Master-Thesis

Konzeption und Einführung von Metriken in den Softwareentwicklungsprozess

Strobel, Stefan

geboren am 28. Juni 1982 in Rodewisch

Studiengang Informatik

Westsächsische Hochschule Zwickau
Fakultät Physikalische Technik/Informatik
Fachgruppe Informatik

Betreuer, Einrichtung: Prof. Dr. W. Golubski, WH Zwickau
Dipl.-Ing. Thorsten Sonnemann, XCOM AG Zwickau

Stefan Strobel
Matrikelnummer: 091079/KT
Auerbacher Str. 106
08147 Crinitzberg

Gegenstand der hier vorgestellten Master-Thesis ist eine Vorgehensweise, wie Metriken in einen vorhandenen Softwareentwicklungsprozess eingebunden werden können. Hierfür wird ein Qualitätsmodell aufgestellt, das den Einsatz von Softwaremetriken zu beschreiben versucht. Der Einsatz der Softwaremetriken beruht auf den definierten Zielen des Unternehmens. Diese Ziele werden zu Beginn dieser Master-Thesis ermittelt. Aufbauend auf den Zielen werden Softwaremetriken erhoben. Dabei werden keine neuen Metriken aufgestellt, sondern mit Hilfe vorhandener Metriken diese Ziele verfolgt.

Ebenso wird nach Werkzeugen gesucht, die definierten Ziele zusammen mit den aufgestellten Metriken in den Softwareentwicklungsprozess zu intergrieren. Ein dauerhaftes Messverfahren mit einer zentralen Bereitstellung von Messergebnissen wird dabei vorgestellt. Die Umsetzung und der Einsatz des Qualitätsmodelles wird ebenfalls besprochen.

Grundlegend wird sich mit verschiedenen Softwaremetriken auseinandergesetzt. Die Diskussionen über die Metriken bilden den Hauptumfang in dieser Master-Thesis.

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung der Master-Thesis in vielfältiger Weise unterstützt haben. Ein besonderes Dankeschön gebührt der Firma XCOM AG. Hier möchte ich mich bei Dipl.-Ing. Thorsten Sonnemann, Dipl.-Ing. Stephan Heinze und der Abteilung SWE III unter Dipl.-Ing. Thorsten Sonnemann besonders bedanken.

Auch möchte ich mich bei allen Familienmitgliedern und Freunden bedanken, die mich zu jeder Zeit unterstützt haben.

Vielen Dank!

Inhaltsverzeichnis

Abkürzungsverzeichnis

ACD	Average Class Dependencies
ACDh	Average Hard wired Class Dependencies
AST	Abstract Syntax Tree
CCN	Cyclomatic Complexity Number
CD	Class Dependencies
CDh	Hard wired Class Dependencies
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
DBC	Duplicated Blocks Complexity
GQM	Goal-Question-Metric
LCOM	Lack of Cohesion in Methods
LOC	Lines of Code
LOD	Law of Demeter
OCP	Open-Closed-Principle
rACD	Reduktionsmetrik zu ACD
rACDh	Reduktionsmetrik zu ACDh
RFC	Response for a Class
SD	Switch Density
SEI	Software Engineering Institut
SM	Survived Mutant
SRP	Single-Responsibility-Principle
TDD	Test Driven Development
TQM	Total Quality Management
VOD	Violation of Demeter

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Aufgabenstellung

Es soll im Zuge dieser Master-Thesis ein Konzept erstellt werden, wie Metriken in einen bereits vorhandenen Softwareentwicklungsprozess eingebunden werden können. Dabei wird auf den aktuellen Softwareentwicklungsprozess eingegangen und nach Möglichkeiten gesucht, diesen zu verbessern bzw. anzupassen, um mit den in dieser Master-Thesis erarbeiteten Metriken arbeiten zu können.

1.2 Arbeitsziel

Ziel der Arbeit ist es, ein Konzept zu erstellen, Metriken in den Softwareentwicklungsprozess der Firma XCOM AG fest zu integrieren und dadurch eine Qualitätssteigerung des Softwareproduktes sicherstellen zu können.

1.3 Vorgehensweise

In Kapitel 2 werden Beispiele für gescheiterte Projekte näher gebracht. Dadurch soll gezeigt werden, dass ein funktionierendes Qualitätsmanagement von essentieller Bedeutung ist.

In Kapitel 3 wird mit Hilfe von Interviews und Umfragen der Ist-Zustand in der Abteilung analysiert und bewertet.

In Kapitel 4 werden mit Hilfe der Zustandsanalysen aus Kapitel 3 die Ziele durch ein Meeting definiert. Diese festgelegten Ziele werden dann durch eine Umfrage ergänzt, indem ausgewählte Entwickler mit ihren Worten beschreiben, wie jedes einzelne Ziel am besten erreicht werden kann. Aus diesen Ergebnissen werden dann entsprechende Metriken in das Qualitätsmodell aufgenommen.

Anschließend wird in Kapitel 5 das Qualitätsmodell unter Berücksichtigung des aktuellen Entwicklungsprozesses integriert. Dabei wird die Umsetzung des Qualitätsmodelles besprochen.

Kapitel 6 stellt eine Fallstudie dar, indem die Integrierung des Qualitätsmodelles in den Entwicklungsprozess untersucht wird. Dabei wird auf einzelne Metriken eingegangen.

Die Arbeit endet in Kapitel 7 mit einer Zusammenfassung und einem Ausblick.

Kapitel 2

Einführung

2.1 Beispiele für Projekte mit fehlerhafter Software

2.1.1 Tippfehler in Befehlssequenz

1988 wurde die russische Raumsonde „Phobos I“ Richtung Mars geschickt. Der Bordcomputer der Sonde war mit einem Programm zum Testen ausgestattet, wobei das Testprogramm im ROM des Bordcomputers gespeichert war. Um es zu entfernen hätte der Bordcomputer ausgebaut werden müssen und somit versah man das Programm mit einem Software-Schloss. Nach 6 Wochen im All schickte man Phobos neue Steuersequenzen, die insgesamt ca. 30 Seiten umfassten. Diese Sequenzen entriegelten das Software-Schloss, weil ein Buchstabe in einer Sequenz vergessen wurde. Dabei kam die Sonde ins trudeln und war verloren. [Tha00b, Seite 15]

2.1.2 Absturz der Ariane 5 Rakete

1996 zerbrach die Ariane 5 Rakete 42 Sekunden nach dem Start, weil ein Unterprogramm des Trägheitsnavigationssystems einen Variablenüberlauf verursachte (64 Bit Float auf 16 Bit Integer). Die Software wurde 1:1 von der Ariane 4 übernommen, da diese bei der Ariane 4 zuverlässig arbeitete und man davon ausging das diese auch auf der Ariane 5 funktioniert. Es war außerdem vorgesehen, bei solch einem Fehler das komplette Trägheitsnavigationssystem abzuschalten. Zwar hatte die Ariane 5 ein Backupsystem, nur war dieses mit selber Hard- und Software bestückt und fiel ebenso aus. Der Bordcomputer empfing anschließend nur noch Diagnosedaten, interpretierte diese aber als Flugdaten. Die Kursänderung waren so gravierend, dass die Ariane 5 auseinander brach. Die Selbstzerstörung wurde eingeleitet.

Dieser kleine Softwarefehler hatte beträchtliche Folgen. Man bezifferte den Schaden auf ca. 370 Millionen US-Dollar. [Tha00b, Seite 15]

2.1.3 Weitere Beispiele

Finanzielle Katastrophen:

- 1987-1993: Integration der kalifornischen Systeme zur Führerschein- und KFZ- Registrierung abgebrochen: 44 Mio US-Dollar
- 1992: Integration des Reservierungssystems SABRE mit anderen Reservierungsystemen abgebrochen: 165 Mio. US-Dollar
- 1997: Entwicklung des Informationssystems SACSS für den Staat Kalifornien abgebrochen: 300 Mio US-Dollar
- 1989-1998: Entwicklung eines Systems zur Unterstützung der Polizeiarbeit in Hamburg abgebrochen: 120 Mio. DM

Terminkatastrophen:

- 1994: Eröffnung des Denver International Airport um 9 Monate verzögert wegen Softwareproblemen im Gepäcktransport-System
- 2003: Toll Collect: Termin u.a. wegen Softwareproblemen auf 18 Monate verschoben, Kosten $>$ 1 Milliarde Euro

Technik-Katastrophen:

- 1985-1987 Therac 25 (Strahlengerät zur Krebsbehandlung): Fehlerhafte Programmierung führt zu Verbrennungen und Todesfällen
- März 1999: Fehlstart einer Titan/Centaur-Rakete wegen falscher Software-Version
- September 1999: Verlust der Sonde „Mars Climate Orbiter“ wegen falscher Einheitenumrechnung

[Pre11, ab Seite 3]

2.1.4 Zusammenfassung

Aus den Beispielen soll deutlich gemacht werden, das Fehler in Softwareprodukten zu erheblichen finanziellen Einbußen führen können. Auch sollte das Motto "Bananensoftware"[1] nicht zum Stil eines Softwarehauses gehören. Wie auch in anderen Industriezweigen, sollte ebenso in der Softwareentwicklung auf Qualität geachtet werden. Dies betrifft den Prozess sowie das Produkt selbst. Aus diesem Grund soll im folgendem Abschnitt das Qualitätsmanagement kurz betrachtet werden.

2.2 Das Qualitätsmanagement

Im folgenden sollen Möglichkeiten für den Weg zu einem funktionierenden Qualitätsmanagement vorgestellt werden. Das Ziel dabei ist es nicht eines dieser Verfahren explizit zu benutzen, da es den Rahmen dieser Arbeit sprengen würde. Es soll nur gezeigt werden welche Möglichkeiten bestehen und umgesetzt werden können. In Kapitel 5 werden die hier vorgestellten Möglichkeiten mit dem Ist- und Sollzustand dieser Abteilung verglichen. Dabei können die erarbeiteten Ansätze eingesetzt bzw. Teile aus diesen verwendet werden, wenn es dies erfordert.

2.2.1 Definition

> Das Qualitätsmanagement umfasst alle Tätigkeiten der Gesamtführungsaufgabe, welche die Qualitätspolitik, Ziele und Verantwortungen festlegen sowie diese durch Mittel wie Qualitätsplanung, Qualitätslenkung, Qualitätssicherung und Qualitätsverbesserung im Rahmen des Qualitätsmanagementsystems verwirklichen. [Geb03]

Diese Definition deckt nicht nur den Bereich der Softwareentwicklung ab, sondern ist allgemeingültig.

Nach [Tha00a] ergeben sich bei der Softwareentwicklung folgende Aufgaben für das Qualitätsmanagement:

- Planen und Dokumentieren des Qualitätssicherungssystems
- Beurteilen von Methoden, Techniken und Werkzeugen für Software

[1] Software, die erst beim Kunden zu einem funktionierenden Produkt anwächst

- Veranlassen von Reviews und Inspektionen
- Durchführen von internen und externen Audits sowie Berichten über das Ergebnis
- Prozessprüfung und -verbesserung
- Teilnahme an Tests und Erstellen eines Protokolls
- Annahme oder Ablehnung von Produkten nach der Durchführung von Tests
- Berichten über Fehler, Mängel und Verbesserungen
- Freigabe von Software, darunter auch Dokumente
- Vertreten von Angelegenheiten der Qualitätssicherung im eigenen Unternehmen, gegenüber Kunden, Anwendern, Lieferanten und Unterauftragnehmern sowie den entsprechenden Repräsentanten in Partnerfirmen

Balzert [Bal98, Seite 355] sagt: "Der Softwareentwicklungsprozess beeinflusst wesentlich die Produktivität und die Qualität der erstellten Produkte." Aus diesem Grund sollen nachfolgend, einige Methoden zur Prozessverbesserung vorgestellt werden, die eine Qualitätssteigerung von Produkt und Prozess versprechen.

2.2.2 Das ISO 9000 Normenwerk

Nach [Bal98, Seite 355] stellt das ISO 9000 Normenwerk allgemeingültige, branchenneutrale Minimalanforderungen an ein Qualitätsmanagementsystem, die da wären:

- vollständig
- dokumentiert
- bekannt
- überprüfbar
- evolutionär
- eingehalten

Im Bereich Softwareentwicklung ist vor allem DIN EN ISO 9000, Teil 3, *Leitfaden für die Anwendung von DIN EN ISO 9001 auf die Entwicklung, Lieferung und Wartung* interessant. Dabei ist DIN EN ISO, Teil 3 keine Norm sondern lediglich ein Leitfaden. [Tha00a, Seite 24]

Ist ein Qualitätsmanagementsystem vollständig nach ISO 9000-3 umgesetzt, dann erfüllt es automatisch die Anforderungen nach ISO 9001. Damit ist es möglich ein Zertifikat zu erhalten, welches die Qualitätseigenschaften eines Unternehmens bescheinigt. [Bal98, Seite 330]

Zusammengefasst enthält die ISO 9001:

- Kundenorientierung
- Verantwortlichkeit der Führung
- Einbeziehung der beteiligten Personen
- Prozessorientierter Ansatz
- Systemorientierter Managementansatz
- Kontinuierliche Verbesserungen
- Sachbezogener Entscheidungsfindungsansatz
- Lieferantenbeziehungen zum gegenseitigen Nutzen

Der ISO 9000 Ansatz gilt als sehr bürokratisch und wird ebenso als prozessorientiert definiert.

2.2.3 Capability Maturity Model Integration (CMMI)

CMMI wurde speziell für den Softwareentwicklungsprozess entwickelt und stellt nicht wie DIN EN ISO 9001 eine Allgemeingültigkeit dar.

Das Software Engineering Institut (SEI) entwickelte 1986 im Auftrag des amerikanischen Verteidigungsministerium einen Fragebogen, mit dem die Leistungsfähigkeit eines Softwareunternehmens bewertet werden sollte. Dieser Fragebogen wurde in ein Referenzmodell umgewandelt. [Bal98, Seite 362] Es entstand das Capability Maturity Model (CMM). Im Jahre 2003 wurde CMM durch CMMI ersetzt, um ein modulares und vereinheitlichtes Model zu erstellen.

Das Modell setzt sich aus den Bereichen Projektmanagement, Prozessmanagement, Support und Entwicklung zusammen. Die einzelnen Bereiche sind dabei noch weiter unterteilbar.

Neben diesen Bereichen existieren 5 Reifegrade in CMMI:

1. Initial - Es gibt keine Anforderungen. Der Prozess kann als chaotisch angesehen werden.
2. Managed - Ein funktionierendes Projektmanagement ist gefordert.
3. Defined - Hier müssen nun alle Teilbereiche miteinander arbeiten. Der Entwicklungsprozess muss einheitlich sein und darf sich nicht, wie beispielsweise in *Managed* von Projekt zu Projekt unterscheiden.
4. Quantitatively Managed - Hier sollen nun anhand von Metriken weitere Optimierungen stattfinden.
5. Optimizing - Noch auftretende Fehler und Probleme werden hier systematisch analysiert und beseitigt.

Der Einsatz von Metriken kommt erst in Stufe 4 zur Geltung. Zwar werden Metriken auch in den darunterliegenden Reifegraden benutzt, jedoch finden diese einen ersten effektiven Einsatz ab dem Reifegrad "Defined". CMMI sieht nicht vor Metriken gezielt einzusetzen, wenn einzelne Prozesse noch nicht aufeinander abgestimmt sind. Zudem geht die Sichtweise dabei stark Richtung Prozessmetriken. Da es pro Reifegrad ca. 2 - 3 Jahre dauern kann bis man in den nächsthöheren Reifegrad gelangt, ist die Umsetzung von CMMI mit erheblichen Aufwand verbunden.

Angesichts dessen, dass das CMMI-Modell in den USA Anwendung findet[2], gewinnt dies in Zukunft sicher noch mehr an Bedeutung. Auch gibt es Ansätze CMMI mit agilen Methoden zu verknüpfen, was gerade für kleinere Unternehmen interessant sein könnte. Dies ist jedoch sehr umstritten und findet momentan mehr Ablehnung als Akzeptanz.

2.2.4 Total Quality Management (TQM)

TQM wird von Balzert [Bal98, Seite 339] wie folgt definiert: "Auf der Mitwirkung aller ihrer Mitglieder basierenden Führungsmethode einer Organisation, die Qualität in den Mittelpunkt stellt und durch Zufriedenheit der Kunden auf langfristigen Geschäftserfolg sowie auf Nutzen für die Mitglieder der Organisation und für die Gesellschaft zielt."

Es werden also sämtliche Organisationen des Prozesses mit eingebunden um die Qualität auf hohem Niveau halten zu können. Somit ist eine komplette Umstrukturierung des Unternehmens notwendig.

[2] in Deutschland bisweilen kaum verbreitet

Auch wird gesagt, dass die Qualität aus Sicht der Kunden das zentrale Ziel ist. Somit entscheidet der Kunde über die Qualität des Produktes. Der Kunde steht demnach mit seinen Bedürfnissen im Mittelpunkt. Es wird ein Qualitätsmanagementsystem gefordert, welches seine Prozesse kennt, ausreichend dokumentiert hat und ständig an neue Erfordernisse anpasst.

Zudem sagt Balzert [Bal98, Seite 353]: "TQM ist mehr eine allgemeine Philosophie, die von jedem Unternehmen unterschiedlich interpretiert werden kann" und "Die ISO 9000 will das Verhältnis Auftraggeber - Lieferant auf eine solide Grundlage stellen, TQM will Qualität im umfassenden Sinne erzielen,..."

Der Softwareentwicklungsprozess Scrum[3] kann als 'Teilphilosophie' von TQM interpretiert werden. Auch mit Scrum wird versucht, den Kunden mit seinen Bedürfnissen in den Mittelpunkt zu stellen. Ebenso wird der Entwicklungsprozess in seiner Komplexität etwas zurückgenommen. Erweitert mit dem Einsatz von Prozess- und Produktmetriken kann sich ein Erhöhen der Qualität ergeben.

2.2.5 Qualitätsmodelle

Mit Hilfe eines Qualitätsmodelles ist es möglich, die Qualität eines Software-Produktes zu beurteilen. Dafür kann auf bestehende Qualitätsmodelle zurückgegriffen bzw. neue Qualitätsmodelle aufgestellt werden.

2.2.5.1 Die ISO 9126

Die ISO 9126 definiert 6 Qualitätsziele. Diese Ziele beziehen sich ausschließlich auf die Produktqualität und sind in weitere Qualitätsmerkmale unterteilt.

Wie in Abbildung 2.1 zu sehen ist, geben die aufgestellten Qualitätsmerkmale nur eine Richtung vor. Die ISO 9126 lässt offen, wie weiter vorgegangen werden kann und bietet lediglich Anhaltspunkte hierfür. Je nachdem welche Ziele ein Unternehmen verfolgt, lassen sich aus diesem Standard einige individuelle Qualitätsmodelle aufstellen.

[3] siehe http://scrum-master.de/Scrum-Glossar/Scrum

Abbildung 2.1: Qualitätsmerkmale nach ISO 9126

2.2.5.2 Der Goal-Question-Metric-Ansatz (GQM)

Mittels GQM lassen sich auf einfache Weise Metriken ermitteln, die die Ziele eines Unternehmens unterstützen. Nach Schneider [Sch06, Seite 69] heißt es: “Man soll nicht das messen, was leicht zu messen ist, sondern das, was man braucht, um seine Verbesserungsziele zu erreichen.” Dies stellt gleichzeitig die Grundidee von GQM dar. GQM geht von Zielen aus und verfeinert diese über Fragen bis hin zu Metriken (Top-Down). Demnach kann mit GQM ein individuelles Qualitätsmodell aufgestellt werden. Produkt- oder Prozessqualität lässt sich mit Hilfe dieser Methode beschreiben.

Nach Schneider [Sch06, Seite 72] gibt es 6 Schritte in GQM nach denen vorgegangen wird:

1. Ziele erheben und schrittweise verfeinern
2. Facettenbeschreibung - Jedes Ziel soll in ähnlicher Form aufgeschrieben werden
3. Fragen aus den Zielen ableiten
4. Metriken und Erhebungsmethoden sammeln
5. Formulare und Fragelisten erstellen
6. Auswertung

Für die Schritte 5 und 6 ist es sinnvoll eine Automatisierung zu verwirklichen. Zur Datenerhebung soll vermieden werden, dies mittels Formularen bewerkstelligen zu müssen. Entsprechende Werkzeuge können dafür eingekauft oder entwickelt werden. Auch soll die Auswertung automatisch geschehen, sodass das Ergebnis bequem abgerufen werden kann. Dies ermöglicht eine Zeiteinsparung und sollte die Produktivität der Entwickler weniger beeinträchtigen.

2.2.5.3 Factor-Criteria-Metrics-Model (FCM-Model)

Das FCM-Modell versucht mittels Qualitätsmerkmalen die Softwarequalität zu beschreiben. Die Qualitätsmerkmale werden dabei weiter in Teilmerkmale bzw. Kriterien aufgeteilt, die dann bewertbar gemacht werden. [Bal98, Seite 258]

Bewertbar machen bedeutet, das an dieser Stelle Metriken die Ausprägung eines Merkmales beschreiben.

Für das Aufstellen der Metriken kann beispielsweise eine Befragung durchgeführt werden oder es kann hierfür ebenfalls GQM zum Einsatz kommen.

Ein FCM-Modell stellt eine gute Grundlage dar, mit individuellen Metriken arbeiten zu können. Das Aufstellen der Qualitätsziele und Merkmale und ebenfalls der Einsatz der Metriken muss jedoch sorgfältig durchgeführt werden.

2.3 Metriken

> Tom DeMarco: "You cannot control what you cannot measure."
> Clerk Maxwell: "To measure is to know."
> Lord Kelvin: "The degree to which you can express something in numbers is the degree to which you really understand it."
> Louise Pasteur: "A science is as mature as its measurement tools."

2.3.1 Definition

> Metriken in der Softwareentwicklung sind der Bestandteil der Softwaremessung, mit deren Hilfe man versucht die Software besser zu verstehen. Dazu dienen die Zahlen. Zahlen helfen, die Zusammensetzung eines komplexen Gebildes, wie ein Softwaresystem, zu begreifen: „Comprehension through Numbers" [Sne10, Seite 6].

Nach Dumke [Dum01, Seite 158] ist die Software-Metrik eine Abstandsfunktion, die Software-Komponenten Zahlen zuordnet. Dabei wird ein mehrdimensionaler Raum aufgespannt, in dem die Metrikwerte einzelne Punkte darstellen. Eine Metrik m muss folgende Bedingung erfüllen um quantitative Aussagen treffen zu können ($a, b, c \in \mathbb{R}$):

- $m(a, a) = 0$ - der Abstand eines Punktes zu sich selbst ist 0

- $m(a, b) = m(b, a)$ - der Abstand von a zu b ist gleich dem von b zu a (Symmetrie)
- $m(a, b) \leq m(a, c) + m(c, b)$ - der Abstand von a zu b ist nicht größer, als die Summe der Abstände a zu c und c zu b (Dreiecksungleichung)

2.3.2 Bewertung

Eine Aussage über den Zustand eines Softwareproduktes allein mittels quantitativer Aussagen ist nur bedingt möglich. Schwierig ist es die "richtigen" Metriken zu finden und diese "richtig" zu interpretieren. Wallmüller [Wal00, Seite 44] sagt zudem: "Ein Maß heißt valide, wenn die Messergebnisse einen eindeutigen und unmittelbaren Rückschluss auf die Ausprägung der Qualitätseigenschaft zulassen."

Es fehlt demnach der Zusammenhang zwischen qualitativen[4] und numerischen Aussagen. An dieser Stelle spielt die Messtheorie eine große Rolle. Hier werden Skalentypen verwendet, die die empirische Bedeutung hinter den Zahlen klassifizieren.

Zu beachten ist, dass nicht direkt messbare Qualitätsaspekte, wie Wartbarkeit oder Erweiterbarkeit, subjektiv interpretiert werden. Entwickler haben unterschiedliche Vorstellungen was Qualität bedeutet. Auch sehen viele Entwickler ihre Produkte als Kunstwerke an und nicht als Produkt ihrer Firma. Dieser subjektive Anteil sollte möglichst gering gehalten werden, um Messungen unterschiedlicher Projekte vergleichen zu können.

2.3.3 Direkte Metriken

Direkte Metriken basieren auf dem zugrunde liegenden Messobjekt (beispielsweise Quelltext). Die Messung erfolgt demnach unabhängig von anderen Metriken und wird direkt ausgewertet.

2.3.4 Indirekte Metriken

Indirekte Metriken werden aus Ergebnissen direkter Metriken berechnet.

[4] siehe Abschnitt 2.2.5.1

2.3.5 Prozessmetriken

Prozessmetriken oder auch Steuermetriken sind im Bereich Softwareentwicklungsprozess nützlich. So lassen sich beispielsweise Kosten eines Projektes vorhersagen oder es lässt sich der Entwicklungsstand eines Projektes ermitteln.

2.3.6 Produktmetriken

Produktmetriken oder auch Vorhersagemetriken lassen sich aus dem Quellcode erheben. Zu ihnen gehören Metriken wie Lines of Code (LOC) oder Response for a Class (RFC).

Produktmetriken lassen sich weiter in dynamische und statische Metriken aufteilen.

Statische Metriken

Statische Metriken werden nicht zur Laufzeit eines Programms erhoben. Auch werden keine Metriken mit Zeitbezug in diese Kategorie berücksichtigt. So lassen sich beispielsweise Wartbarkeit, Architektur oder auch die Testbarkeit über solche statischen Analysen ermitteln.

Dynamische Metriken

Dynamische Metriken hingegen besitzen einen Zeitbezug. Beispielsweise lassen sich Laufzeiten eines Programms ermitteln oder Fehler pro Zeitintervall festhalten. Da die Messung im allgemeinen etwas schwieriger ist, wird diese auch weniger genutzt als die statische Analyse.

Der Einsatz von dynamischen Metriken kann weitere Fehler aufdecken wozu statische Metriken nicht in der Lage sind. Aus diesem Grund ist der Einsatz von dynamischen Metriken ebenso erforderlich, wie der Einsatz von statischen Metriken.

2.3.7 Weitere Kategorien

Die statischen und dynamischen Metriken lassen sich noch weiter unterteilen. Unter Produktmetriken sind auch Metriken der Objektorientierung wiederzufinden wie Vererbung, Kapselung, Kopplung, Kohäsion und Polymorphismus.

Weitere Unterteilungen:

- Umfangsmetriken
- Strukturmetriken

- Objektorientierte Metriken

2.3.8 Grundlagen der Messtheorie

Eine Metrik (oder auch Maß) beschreibt die Abbildung eines empirischen Relationensystems auf ein formales Relationensystem. Für ein besseres Verständnis soll dies anhand der Komplexität von Quelltext erläutert werden.

Eine empirische Relation für die Komplexität wäre beispielsweise:

Klasse A **ist komplexer als** Klasse B.

Eine formale Repräsentation dieses Sachverhaltes kann mit den Werten der Komplexität für Klasse A und Klasse B beschrieben werden. Angenommen der Komplexitätswert für Klasse A beträgt 25 und der Komplexitätswert der Klasse B beträgt 20, so ist die formale Repräsentation $25 > 20$ als korrekt anzusehen, was die empirische Relationen implizit akzeptiert.

Wie dabei die Komplexität gemessen wird, definiert die Metrik für Komplexität. Das Repräsentationsproblem beschreibt dabei nur, ob alle definierten Relationen auch formal gültig sind.

Ein empirisches Relationensystem besteht aus direkten oder indirekten Entitäten aus der realen Umgebung. Zwischen diesen Entitäten gibt es eine oder mehrere Relationen. Das formale Relationensystem besteht (nur) aus Zahlen und deren Relationen untereinander. Eine Metrik ist gültig, wenn die Repräsentationsbedingungen erfüllt sind. Dies bedeutet, dass zu jedem Element und jeder Relation im empirischen Relationensystem eine formale Repräsentation vorhanden ist. [Ben04, Seite 12ff]

Werden diese Eigenschaften berücksichtigt, so beschreibt die empirische und formale Repräsentation zusammen mit dem Maß oder der Metrik eine Skala. Jede Skala beschreibt dabei die Zuordnung der Messwerte und zeigt mögliche Operationen mit den Messwerten. Tabelle 2.1 zeigt eine Übersicht der wichtigsten Skalentypen.

Betrachtet man indirekte Metriken, so sind erlaubte Operationen nur auf den zugrunde liegenden Metriken abzuleiten bzw. erlaubt.

Skalentyp	Operationen	Transformation
Nominalskala	$=, \neq$	ein Element auf ein anderes
Ordinalskala	$=, \neq, <, >$	g: streng monotone Funktion
Intervalskala	$=, \neq, <, >, +, -$	$g(x) = ax + b, a > 0$
Rationalskala	$=, \neq, <, >, +, -, *, /$	$g(x) = ax, a > 0$
Absolutskala	$=, \neq, <, >, +, -, *, /$	$g(x) = x$

Tabelle 2.1: Skalentypen (vgl. u.a. [Zus98, Seite 131ff], [Dum96, Seite 237ff])

2.3.9 Validierung von Metriken

Für einen erfolgreichen Einsatz von Metriken müssen diese validiert werden. Es wird dabei zwischen interner, sowie externer Validierung unterschieden. [Zus98, Seite 418]

Die interne Validierung beschreibt, ob eine Metrik konform zur Messtheorie ist. Die externe Validierung beschreibt, ob ein festgelegtes Kriterium und die zugrunde liegende Fragestellung beantwortet werden kann. Für die externe Validierung ist hierfür eine statistische Untersuchung erforderlich, wobei hingegen die interne Validierung zwei Schritte benötigt:

1. Überprüfen des Skalentyps
2. Prüfen der Repräsentationsbedingung

Für die Prüfung der Repräsentationsbedingung ist ein Anforderungskatalog notwendig. Dieser muss vollständig sein und die Realität möglichst genau abbilden. Das Repräsentationsproblem beschreibt Relationen zwischen einem empirischen relationalen System **A** und einem dazugehörigem relationalem System (B). Hierzu müssen Messstrukturen entwickelt werden, die als Axiome[5] bezeichnet werden. [Dum96, Seite 236]

[5] auch Grundannahmen, Postulate oder Bedingungen genannt

Kapitel 3

Anforderungen

3.1 Zustandsanalyse

Im folgendem Kapitel soll eine durchgeführte Zustandsanalyse zusammengefasst werden. Hierzu wurden Entwickler und Chefentwickler zu einer Umfrage eingeladen. Zusätzlich wurden mit den Chefentwicklern und Projektmanagern einzelne Interviews durchgeführt.

Das Ziel war es, einen ersten Eindruck über den Wissensstand vorhandener Metriken zu bekommen und die Bestimmung erster Qualitätsziele.

3.1.1 Umfrage an die Entwickler

3.1.1.1 Inhalt der Fragen

Die Umfrage wurde elektronisch durchgeführt und es wurde eine Woche Zeit gegeben alle Fragen zu beantworten.

In Abbildung 3.1 ist die Teilnahme an der Umfrage dargestellt. Insgesamt wurden 22 Entwickler zur Umfrage eingeladen. Somit ist die Resonanz zufriedenstellend und es lässt sich ein erster Eindruck ermitteln, inwieweit die Entwickler mit dem Thema Softwaremetriken vertraut sind.

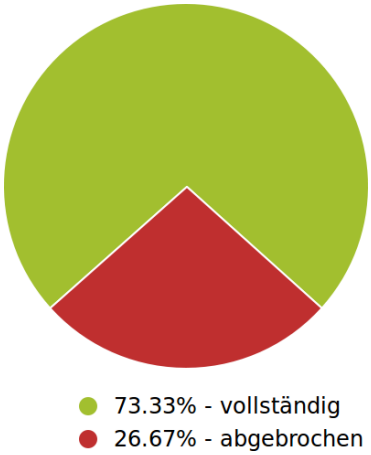

Abbildung 3.1: Teilnahme der Entwicklerumfrage

Frage 1 - Hast Du dich schon mal mit dem Thema Softwaremetriken auseinandergesetzt?

Da in der Abteilung schon mit Softwaremetriken gearbeitet wird, dies jedoch jeder Entwickler für sich entscheiden kann, sollte diese Frage ermitteln, wie verbreitet das Grundverständnis zu dem Thema ist.

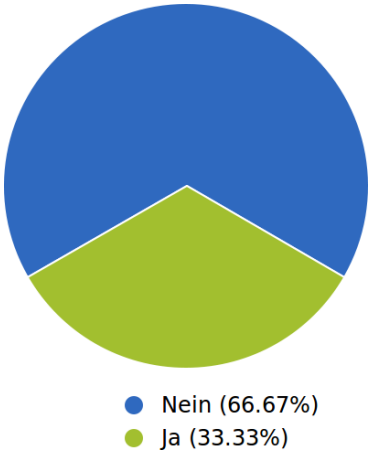

Abbildung 3.2: Vorhandenes Wissen über Softwaremetriken

Wie in Abbildung 3.2 zu erkennen ist, haben sich 33% der Teilnehmer bereits mit Metriken beschäftigt. Somit lässt sich zusammenfassen, dass die bisherigen Anstrengungen, Metriken in der Abteilung zu etablieren, mehr oder weniger ignoriert wurden. Dies lässt sich aus Mangel an Informationsfluss, mangelndem Interesse oder am Nichtvorhandensein von Schulungen begründen.

Fragen 2-4 - Kenntnisse über Metriken

Diese Fragen konnten nur beantwortet werden, wenn Frage 1 mit *Ja* beantwortet wurde. Aus den Antworten lässt sich schließen, dass hier nur Grundwissen vorhanden ist. Da ein Grundwissen nicht ausreichend ist, um effektiv mit Metriken arbeiten zu können, sollte hier eine Verbesserung stattfinden.

Frage 5 - Nutzt Du Sonar (sonarsource.org) in unserer Produktionsumgebung?

Wie in Abbildung 3.3 zu sehen, gehört die Instanz Sonar[1] nicht zum Standardwerkzeug in der Abteilung. Sonar ist eine Plattform, die die Qualität von Java-Quellcode untersucht. Dabei werden die Ergebnisse visuell dargestellt. Jeder in der Abteilung hat Zugriff darauf (per Webinterface) und kann den Zustand aller Projekte einsehen.

Es wurden Coding-Rules für die Abteilung festgelegt, die das Layout des Quellcodes vereinheitlichen sollen. Zudem werden mit Hilfe von FindBugs[2] und PMD[3] Fehler im Quellcode aufgedeckt. Auch werden grundlegende Metriken unterstützt, die über einen längeren Zeitraum verglichen werden können. Auch lässt sich mit Hilfe von Plugins die Funktionalität von Sonar erweitern.

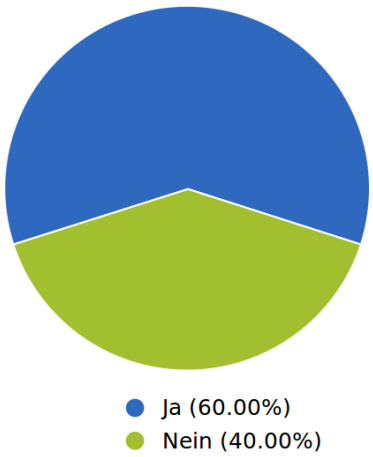

Abbildung 3.3: Nutzung von Sonar

[1] siehe http://www.sonarsource.org
[2] siehe http://findbugs.sourceforge.net/
[3] siehe http://pmd.sourceforge.net/

Fragen 6-8 - Kenntnisse über Sonar

Hier wurden allgemeine Kenntnisse von Sonar abgefragt. Ziel war es herauszufinden, welche Funktionen genutzt werden und ob es Probleme bereitet die verfügbaren Informationen zu interpretieren. Ergebnis ist, dass bisweilen nur Wert auf die Funktion "Rules Compliance mit Violation Drilldown" gelegt wird. Diese Funktion beinhaltet FindBugs, PMD und Checkstyle. Sie zeigt Fehler am Quellcode an und überwacht die Einhaltung der festgelegten Coding-Rules (Checkstyle).

Fragen 9-12 - Qualitätseigenschaften

Hier wurden Meinungen über Qualitätseigenschaften von Softwareprodukten abgefragt. Es sollte ebenso ein Statement zum aktuellen Softwareentwicklungsprozess abgegeben werden.

Auffallend dahingehend ist, dass Entwickler für den Support direkt eingesetzt werden. Den Aussagen zufolge ist dies oft störend. In diesem Zusammenhang wurde auch das Bemühen um Qualität angesprochen, nur leider steht dafür meistens keine Zeit zur Verfügung.

Fragen 13-14 - Entwicklungswerkzeuge

Diese Fragen dienten für einen Überblick der eingesetzten Werkzeuge und mit welchen Programmiersprachen gearbeitet wird. Eclipse und IntelliJ können hierbei als Standardwerkzeuge in der Abteilung betrachtet werden. Ebenso kann Java als Hauptsprache angesehen werden. Auch wurde Java für zukünftige Projekte strategisch festgelegt.

3.1.1.2 Ergebnis

Die Umfrage lässt einen ersten Blick auf die Wissensverteilung von Softwaremetriken in dieser Abteilung zu. Da bereits seit ca. einem Jahr mit Metriken gearbeitet wird, ist das Ergebnis eher ernüchternd. Es wurde mit Sonar ein System integriert, welches den Einstieg in Softwaremetriken vereinfachen soll, jedoch findet das System bei den Entwicklern kaum Anwendung. Gründe hierfür können sein, dass den Entwicklern nicht genügend Zeit zur Verfügung steht, weniger auf Qualität geachtet wird oder Kenntnisse nicht vorhanden sind. Da eine Schulung bezüglich dessen nie stattgefunden hat und nicht davon ausgegangen werden kann, dass jeder Entwickler seine Freizeit für Weiterbildungen nutzt, muss in der Form *Bildung -> Verständnis -> Anwendung* ein Weg gefunden werden, die Akzeptanz dessen zu erhöhen. Auch muss festgelegt werden, dass der Einsatz von Metriken eine gewisse Priorität bekommt. Es sollte jedem die Möglichkeit gegeben werden, Metriken

anzuwenden, unabhängig von Zeit- und Kostenfaktoren. Zudem soll nicht das Gefühl entstehen, durch Einsatz von Metriken unter Beobachtung zu stehen.

3.1.2 Interviews mit den Chefentwicklern

3.1.2.1 Inhalt der Fragen

Die Chefentwickler wurden ebenfalls zu der Entwicklerumfrage eingeladen, da diese ebenfalls an der Entwicklung von Software beteiligt sind. Mit den anschließenden Interviews sollte jedoch ein tiefgründigeres Wissen bezüglich aktuellem Entwicklungsprozess erfragt werden. Es sollte sich ein Bild ergeben, inwiefern Entwickler und Chefentwickler zusammenarbeiten, ob der Einsatz von Projektmanagementwerkzeugen erfolgt, wie neue Projekte den Weg zur Entwicklung beschreiten und ob es bereits Anstrengungen gegeben hat, gewisse Standards im aktuellen Softwareentwicklungsprozess zu etablieren.

3.1.2.2 Ergebnis

Der Entwicklungsprozess

Aus den Antworten lässt sich zunächst ableiten, dass kein Entwicklungsprozess definiert ist. Ebenso wenig sind Standards festgelegt mit denen gearbeitet werden kann.

Der Einstieg und die Prozessfortschreitung erfolgt aus Erfahrungen heraus. Zwar wurden bereits Werkzeuge für Projektmanagement eingesetzt (beispielsweise MS-Projekt), jedoch erfolgt dies je nach Ermessen und findet eher selten Einsatz. Bemängelt wurde auch, dass fehlendes Fachwissen kundenseitig sehr oft zu verzeichnen ist. Dadurch entstehen fehlerhafte, grobe und unvollständige Spezifikationen, mit denen sich nicht arbeiten und planen lässt.

Auch sind Festpreisvereinbarungen die Regel, sodass bei fehlerhaften Spezifikationen meist kein Gewinn erzielt werden kann. Dabei ist zu beachten, dass für Pilotprojekte das Risiko eingegangen werden kann[4]. Hierfür sollte der Projektverlauf in jedem Fall dokumentiert werden (Gesprächsprotokolle, Verzögerungen, Änderungen) um Probleme im Nachhinein analysieren zu können, was zukünftigen Pilotprojekten zu Gute kommen kann. Dies soll jedoch nicht weiter in dieser Arbeit untersucht werden.

Ein weiteres Problem sind die hohen Reibungskräfte zwischen den Projekten. Jeder Entwickler arbeitet nicht nur an einem Projekt, sondern betreut auch anderweitig weitere

[4] zum Zweck der Neukundengewinnung

Projekte. Sei es Support, sei es Fehlerbehebungen oder sei es eine Parallelentwicklung eines anderen Projektes. Dadurch gibt es zu viele Unterbrechungen, die sehr viel Zeit kosten können und somit Ressourcen unbemerkt verschlingen.

Legt man den Fokus mehr auf den Entwickler, so erkennt man, dass hier bereits einiges getan wurde. Im Vergleich zu anderen Abteilungen wird hier bereits mit GIT[5] gearbeitet. Auch wurde erfolgreich versucht, Arbeit den Entwicklern abzunehmen. Beispielsweise können Build, Deployement, Integrationstests und Releases per Knopfdruck oder automatisch für ein Projekt durchgeführt werden[6]. Auch wird versucht, den Einstieg zu Metriken so leicht wie möglich zu gestalten, indem Sonar eingesetzt wird.

Für interne Projekte wurde bereits mit Scrum[7] versucht, mehr Flexibilität und Ordnung in den Entwicklungsprozess zu bringen. Dieser konnte sich jedoch mangels Interesse und Beteiligung nicht etablieren. Ebenso wurde versucht, die testgetriebene Entwicklung (TDD) als Werkzeug in der Abteilung zu etablieren. Dies kann jedoch auch als gescheitert betrachtet werden, da hier nicht konsequent damit umgegangen wird.

Es gibt demnach keinen roten Faden im gesamten Entwicklungsprozess. Auch wurden Mängel an Bildungs-, Lern- und Meinungsaustausch geäußert.

Zusammenarbeit mit Kunden

Die Zusammenarbeit erfolgt über ein Ticketsystem [8], mittels Konzeptaustausch und Präsentationen. Ebenso wird mittels Lasten- und Pflichtenheft ein Konzeptionsprozess durchgeführt. Über das Ticketsystem werden Fehler und Features festgehalten. Mantis ist jedoch weit entfernt von einem Projektmanagementsystem. Es fehlt der Bezug zu Meilensteinen, Ressourcenverbrauch und Aufgabenverteilung. Es gibt ein internes Wiki, welches für die Projektplanung verwendet wird. Darin sind grob Anforderungen festgehalten, welche Probleme noch zu klären sind und wie weiter vorgegangen werden könnte. Zudem sind teilweise Meilensteine definiert. Aussagen über Ist-Zustand lassen sich jedoch nur erschwert treffen, da die Informationen sehr verstreut sind. Ein zentrales Werkzeug, welches die geforderten Funktionen zur Verfügung stellt, wäre vorteilhaft.

Fehlerdokumentation

Im Bereich Fehlerdokumentation wird nichts getan. Wie bereits angesprochen ist das Festhalten des Projektverlaufes wünschenswert. Es befinden sich auch Projekte in der

[5] siehe dazu http://git-scm.com/
[6] Continuous Integration - Kent Beck, Martin Fowler
[7] agiler Softwareentwicklungsprozess
[8] siehe dazu http://www.mantisbt.org/ - für Kunden, als auch Entwickler zugänglich

Wartungsphase. Hier werden Fehler über das Ticketsystem bekannt gegeben. Diese werden aber nirgends festgehalten bzw. dokumentiert.

Da der Transaktionsserver TRISTAN aus mehreren Modulen besteht, gibt es auch hier die Möglichkeit einer detaillierten Fehlerdokumentation. Es lässt sich dadurch im Nachhinein feststellen, welche Änderungen in Modulen zu Fehlern in anderen Modulen führten. Nach Aussage eines Chefentwicklers: “Die Verantwortlichkeit eines jeden Einzelnen fehlt. Es wird nicht genügend nachgeforscht welche Auswirkungen die aktuellen Änderungen auf andere Projekte haben können.” Beispielsweise kann eine Anpassung von Datenbanktabellen andere Module “beschädigen”. Diese werden aber frühstens im nächsten Release erkannt. Es ist jedoch schwierig, dies automatisch anhand von Metriken festzustellen. Integrationstests können an dieser Stelle sehr hilfreich sein. Damit lassen sich vor Auslieferungen an Kunden, Fehler eingrenzen.

Betrachtungswinkel

Da es schwierig ist allem gerecht zu werden, wurde der Betrachtungswinkel auf die Qualität von Quellcode in den Vordergrund gestellt. So ist der Einsatz von Prozessmetriken vorerst gescheitert.

3.1.3 Interviews mit den Projektmanagern

3.1.3.1 Inhalt der Fragen

Im Vordergrund stand der Entwicklungsprozess und dieser sollte so detailliert wie möglich beschrieben werden.

3.1.3.2 Ergebnis

Der Entwicklungsprozess

Es wurde hier gesagt, dass ein geregelter Prozess nicht möglich ist, da die Anforderungen zumeist unvollständig sind. Zudem wird in der Regel ein Festpreis vereinbart. Daraus wird ersichtlich, dass der Konzeptionsprozess vernachlässigt wird. Zudem wurde betont, dass dies aus Kundensicht geschieht und dieser sich dabei auf den Dienstleister verlässt. Festpreisvereinbarungen und unvollständige Spezifikationen können zudem zur Preispolitik des Kunden gehören. Durchaus gibt es Ansätze aus der Abteilung dem entgegen zu wirken. So werden Entwürfe bzw. Layouts von Oberflächen mit dem Kunden detailliert abgestimmt, weil sich auch hier in der Vergangenheit auf den Dienstleister verlassen

wurde. Auch ist eine weitere Komponentisierung des TRISTAN geplant, wodurch noch flexibler auf Kundenwünsche eingegangen werden kann. Verknüpft mit einer Reihe von Tests, soll sicher gestellt werden, dass der Austausch von Komponenten keine fachlichen und technischen Fehler in anderen Komponenten nach sich zieht. Ebenso wurde betont, dass die meisten Projekte positiv abgeschlossen werden und Kunden durchaus zufrieden mit den Dienstleistungen sind.

Der Kundenumgang

Der Kundenumgang ist noch zu wenig. Wünschenswert wäre ein stetiger Kontakt mit Kunde und Entwicklerabteilung. Da es dafür an Personal mangelt, lässt sich dies im Augenblick so nicht verwirklichen, was der Konzeptionsphase jedoch zu Gute kommen würde.

3.2 Zusammenfassung

Zusammengefasst lässt sich sagen, dass mehr Qualität am Quellcode geschaffen werden soll. Mehr Qualität bedeutet, dass die Idee dahinter gelebt werden muss. Dies erfordert jedoch eine hohe Akzeptanz, die im Moment noch nicht gegeben ist. Das Hauptziel bzw. die Hauptziele sind also mehr Produktqualität verknüpft mit einer hohen Akzeptanz.

Da kein Entwicklungsprozess fest definiert ist, soll zunächst der Einsatz von Produktmetriken verstärkt werden. Hierzu wird im folgenden Kapitel ein Qualitätsmodell aufgestellt. Dabei soll dieses Qualitätsmodell den Entwickler im gesamten Entwicklungsprozess begleiten, sodass an dieser Stelle ein Qualitätsmodell ausreichend ist. Da mit Sonar bereits ein Grundstein gelegt wurde, auf Produktmetriken zu reagieren, soll auch im Zuge dieser Master-Thesis auf Sonar nicht verzichtet werden.

Der Einsatz dieses Qualitätsmodelles und die Umsetzung in Sonar soll in Kapitel 5 diskutiert werden.

Kapitel 4

Aufstellung des Qualitätsmodelles

4.1 Vorgehensweise

Das Aufstellen eines Qualitätsmodelles soll mit Hilfe von FCM geschehen. Um geeignete Ziele für die Abteilung zu bestimmen, wurden dazu die Chefentwickler befragt. Hierzu wurde ein Meeting veranlasst, um Unstimmigkeiten und subjektive Sichtweisen der Zieldefinition zu kompensieren. Es wurden dabei verschiedene Ansätze für ein Modell vorgeschlagen, um Diskussionsgrundlagen zur Verfügung zu stellen. Ebenso ist gewünscht, dass das Qualitätsmodell akzeptiert und später eingesetzt wird.

Zudem sagt Sneed [Sne10, Seite 282]: "Die Menschen können Programme am besten warten, die in einem Stil verfasst sind, mit dem sie vertraut sind." Daher ist es sinnvoll, ein eigenes Qualitätsmodell aufzustellen.

4.1.1 Metrikauswahl

Die Metrikauswahl soll mit Hilfe einer Befragung der Entwickler durchgeführt werden. Es soll von den Entwicklern aufgezeigt werden welche Schwierigkeiten bestehen die definierten Ziele umzusetzen. Anhand dieser Ergebnisse sollen Metriken ausgewählt werden, die diese Probleme am besten beschreiben können. Bei der Auswahl der Metriken soll ebenfalls die Verfügbarkeit in Sonar untersucht werden.

4.2 Übersicht Qualitätsmodell

Wie in Abbildung 4.1 zu erkennen, wurde als oberstes Ziel die Erweiterbarkeit festgelegt. Diese setzt sich aus der Testabdeckung von Unittests, der Testabdeckung von Integrationstests sowie der Testbarkeit zusammen.

Testabdeckung Unittests

Mit der Testabdeckung von Unittests soll entschieden werden, ob Unit- und/oder Integrationstests implementiert werden müssen. Im Zusammenhang mit der Testbarkeit einer Klasse soll dann entschieden werden, welche Klassen sich besonders dafür eignen, mit Tests zu erweitern.

Testbarkeit

Ist eine gute Testbarkeit gegeben, so lassen sich einzelne Komponenten besser testen. Mit der Testbarkeit soll wiedergegeben werden, wie aufwändig bzw. schwierig es ist, eine bestehende Klasse mit Unittests zu erweitern. Der Testaufwand und die Testbarkeit stehen hierbei in einem direkten Zusammenhang.

Testabdeckung Integrationstests

IT-Tests (Integrationstests) fassen mehrere Komponenten zusammen und testen das Verhalten untereinander. Hierzu soll ebenfalls die Testabdeckung auf Klassenebene gemessen und dargestellt werden. Fällt die Testabdeckung von IT-Tests mit der Sicht auf eine Klasse hoch aus, kann die Wahrscheinlichkeit geringer sein Fehler unentdeckt zu implementieren.

Zusammenspiel der Ziele

Mit dem Erweitern von Funktionalität bestehender Klassen, sind Aussagen über die bestehende Testabdeckung sehr wichtig. Hat man eine geringe Testabdeckung im Bereich Unittests sowie IT-Tests, so besteht eine höhere Gefahr, Fehler zu implementieren oder das Verhalten der Klasse unbemerkt zu ändern. Ist zusätzlich eine schlechte Testbarkeit der zu erweiternden Klasse gegeben, so kann damit gerechnet werden, dass ebenso unbemerkt Fehler implementiert werden.

Ist aber eine geringe Testabdeckung der Unittests, sowie eine schlechte Testbarkeit gegeben, jedoch die Testabdeckung der IT-Tests sehr hoch, so kann mit weniger Angst diese Klasse bearbeitet werden.

Mit diesen Zahlen soll entschieden werden, ob zunächst Unit- und/oder IT-Tests implementiert werden müssen, bevor die Klasse erweitert oder verändert wird. Dies soll die Fehlerrate verringern und gleichzeitig die Akzeptanz, sauberen Quellcode zu schreiben, erhöhen.

Diese drei festgelegten Ziele sollen zusätzlich mit Wichtungen angepasst werden können, sodass jedes Projekt optimal auf dieses Modell angepasst werden kann. Dies ist ebenso eine Forderung, die aus dem Meeting entstanden ist.

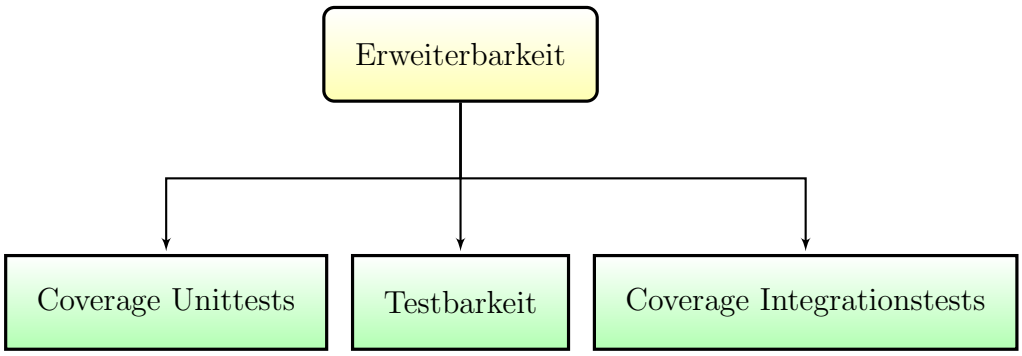

Abbildung 4.1: Qualitätsmodell Erweiterbarkeit

4.3 Entwicklerbefragung

Wie bereits aus der Umfrage bekannt ist, wurde die testgetriebene Entwicklung in dieser Abteilung versucht einzuführen. Daher sind Unittests den Entwicklern kein Fremdwort. Ebenso ist gut aus dem Modell erkennbar, dass das Testen nach wie vor eine große Rolle in dieser Abteilung spielt.

TDD besagt, dass zuerst ein Testfall vor der Implementierung erstellt wird. Hierbei wird sichergestellt, dass jede Codezeile getestet ist. Stimmt diese Aussage, so sollten näherungsweise 100 % Code-Coverage erreicht werden. TDD fördert die Testbarkeit einer Klasse, da stets der Test die Implementierung vorantreibt. Man ist bestrebt, den Test so einfach wie möglich zu gestalten. Ebenso möchte man sich iterativ auf einfache Weise durch diesen Prozess durcharbeiten. Somit können Klassen entstehen, die automatisch eine hohe Testbarkeit besitzen.

Damit also eine hohe Testabdeckung erreicht werden kann, muss die Testbarkeit auf einem entsprechend einfachen Niveau gehalten werden. Die Testabdeckung von aktiven Projekten ist jedoch vergleichsweise gering.

Zur Beurteilung der Testbarkeit einzelner Klassen, wurden Entwickler befragt. Aus den Erkenntnissen sollen anschließend Metriken ausgearbeitet werden.

Es wird angenommen, dass die Testbarkeit einen erheblichen Einfluss auf die Testabdeckung von Unittests hat. Fällt die Testbarkeit einer Klasse gering aus, so muss man mit keinen oder nur wenigen Unittests rechnen.

Das zu erreichende Ziel ist demzufolge eine hohe Testabdeckung der Unittests. Dieses Ziel wurde so in der Umfrage dem Entwickler dargestellt.

Es wurde bewusst nicht der GQM-Ansatz gewählt, da mit der Umfrage explizit testkritische Quellcodezeilen aufgedeckt werden sollten. Hierzu wurden Entwickler befragt, die mit ihren Projekten vertraut sind und ebenfalls Erfahrungen mit der Implementierung von Unittests haben. Da es das Ziel ist, Metriken zur *Testbarkeit* zu finden, können sich durch diese Methode bessere Ergebnisse einstellen.

4.3.1 Beurteilung der Testbarkeit

Aus der Umfrage ergaben sich diverse Erkenntnisse, die kurz vorgestellt werden. Im darauf folgenden Abschnitt sollen diese Erkenntnisse mit Metriken bewertet werden, die dann in das Qualitätsmodell aufgenommen werden.

Zusammengefasst stellen sich folgende Probleme, die aus der Umfrage entstanden sind:

- Initialisierungen von Klassenvariablen (innerhalb des Konstruktors oder auch bereits bei Deklarationen)
- Initialisierungsblöcke
- Zugriffe auf Parameter
- Doppelter Quellcode
- Kohäsion
- Polymorphismus und Kapselung

4.3.1.1 Initialisierung von Klassenvariablen

Bei der Umfrage wurden Klassen gezeigt, die unter anderem im Konstruktor ein Initialisieren von Klassenvariablen durchführten. Diese Art der Initialisierung deutete auf ein Problem der Testbarkeit hin.

In Abbildung 4.2 wird ein Beispiel hierzu vorgestellt. Es werden vier Klassenvariablen im Konstruktor initialisiert.

```
1     public ContextProcessor(final StatementProcessor spIn) {
2       contextDAO = new ContextDAO(spIn);
3       deleteContext = new DeleteContext(spIn);
4       keepReferentialIntegrity = new ReferentialIntegrity(spIn);
5       timeProvider = new SystemTimeProvider();
6     }
```

Abbildung 4.2: Der Konstruktor der Klasse ContextProcessor

Der Vollständige Quellcode der Klasse ContextProcessor ist im Anhang A.1.1 zu finden.

Diese Form eines Konstruktors kann durchaus testkritisch sein. Möchte man die Klasse ContextProcessor testen, so muss diese instanziiert werden. Bei der Instanziierung wird der Konstruktor aufgerufen, der wie im Beispiel die entsprechenden Klassenvariablen initialisiert. Man hat keinen Einfluss auf die Klassenvariablen der Klasse. Der Einsatz von Mockobjekten[1] ist an dieser Stelle nicht möglich, da der new - Operator dies verhindert.

[1] Platzhalter für echte Objekte

Möchte man Mockobjekte einsetzen, so müssen entweder über Setter-Methoden die Klassenvariablen nach der Instanziierung ausgeführt werden, oder man implementiert Initialisierungsblöcke im Konstruktor der zu testenden Klasse. Diese Initialisierungsblöcke müssen dann als Package-Private deklariert werden, damit diese im Test überschrieben werden können.

Der Aufwand, die Klasse testbar zu machen, steigt dabei. Ebenso können Initialisierungsblöcke noch weitaus mehr Arbeit verrichten, als wie der aufgeführte Konstruktor (Abbildung 4.2) darstellt. Setter-Methoden und Initialisierungsblöcke werden **nur** für den Zweck implementiert, die Klasse testbar zu machen bzw. diese von ihrer Umgebung zu isolieren. Dies muss jedoch nicht immer der Fall sein. Die gezeigten Beispiele zeigten jedoch solche Vorgehensweisen.

Ausnahmen müssen jedoch berücksichtigt werden. So müssen Klassen, die lediglich Informationen in einem System transportieren, nicht zwingend testkritisch sein. So kann beispielsweise eine Klasse vom Typ Konto durchaus innerhalb einer Klasse instanziiert werden. Auch wird diese Klasse Methoden anbieten um beispielsweise dem Konto eine Kontonummer bekannt zu machen bzw. danach zu fragen. Solche Objekte können ebenfalls in einem System durchgereicht werden, um andere Klassen mit diesen Informationen zu versorgen. Solche Typen von Klassen dienen lediglich der Datenhaltung und dem Transport der Informationen. Instanziierungen solcher Typen sollten nicht als testkritisch betrachtet werden.

4.3.1.2 Initialisierungsblöcke

Das Verhalten von Initialisierungsblöcken wurde bereits im Abschnitt 4.3.1.1 Initialisierung von Klassenvariablen kurz besprochen. Aus der Entwicklerumfrage wurde mehrfach ersichtlich, dass ein Testen der Klasse erst dann möglich war, wenn Initialisierungsblöcke implementiert wurden, die dann in einen Test überschrieben werden konnten.

In Abbildung 4.3 ist diese Vorgehensweise dargestellt.

In Zeile 18 wird die Init-Methode der Klasse DatabaseOrderStorage aufgerufen. In dieser Methode werden über Setter-Methoden die Klassenvariablen initialisiert. Diese Variante erlaubt es, in einem Unittest Klassenvariablen mit Mockobjekten zu simulieren. Die Init-Methode hat demnach nur den Sinn, in einem Test die Klasse von ihrer Umgebung zu isolieren. Konkret bedeutet dies, dass Abhängigkeiten im Test mit Hilfe der Init-Methode abgeschnitten werden können.

Der Konstruktor der Klasse DatabaseOrderStorage initialisiert 11 Klassenvariablen. Damit ein vollständiges Testen möglich wird, sind insgesamt 11 Mockobjekte notwendig. Zusätzlich muss eine neue Unterklasse von DatabaseOrderStorage erzeugt werden, wo die Init-Methode überschrieben werden muss. Die eigentliche Init-Methode kann nun nicht mehr getestet werden. Dies mindert das Testcoverage. Ebenso ist das Erzeugen einer Instanz von DatabaseOrderStorage aufwändiger.

```
1     private UploadBlockingPeriodDAO uploadBlockingPeriodDAO;
2     private ForeignBankConnectionDAO foreignBankConnectionDAO;
3     private ForeignDownloadDAO foreignDownloadDAO;
4     private OrderConverter orderConverter;
5     private Configuration config;
6     private InventoryProvider inventoryProvider;
7     private UpdateInventory updateInventory;
8
9
10    public DatabaseOrderStorage(
11      final SubSysFactory subSysFactoryParam,
12      final StatementProcessor statementProcessorParam,
13      final Connection connectionParam,
14      final ConverterFactory converterFactory) throws SQLException {
15        connection = connectionParam;
16        statementProcessor = statementProcessorParam;
17        subSysFactory = subSysFactoryParam;
18        init(converterFactory);
19        stopWatch.startNew();
20    }
```

Abbildung 4.3: Konstruktor von DatabaseOrderStorage

Die Init-Methode ist im Anhang A.1.2 zu finden. Ebenso wird demonstriert, wie das Testen dieser Klasse funktioniert.

4.3.1.3 Anzahl der Bedingungen

In der Umfrage wurden vermehrt Klassen gezeigt, die eine hohe Anzahl von Bedingungen enthielten. Dazu wurde gesagt, dass sich solche Klassen schwer testen lassen und der Aufwand eine komplette Testabdeckung umzusetzen mit überproportionalem Aufwand verbunden ist.

Abbildung 4.4 stellt einen Ausschnitt einer komplexeren Methode dar. Prinzipiell ist die Methode testbar. Alle benötigten Informationen werden über die Parameter der Methode zur Verfügung gestellt. In einem Test ist es möglich sämtliche Konstellationen nachzubilden.

Das Erreichen von 100% Testcoverage setzt voraus, dass jede Bedingung geprüft werden muss. So setzt sich Testcoverage aus Line-Coverage und Branch-Coverage zusammen. Für das Konstrukt von Zeile 6 bis Zeile 13 sind insgesamt sechs Tests notwendig. Ein siebenter Test ist nötig, der für keine der Bedingungen aus Zeile 6 zutreffend ist.

```
1         SecurityProfile to = new SecurityProfile();
2         if (null == method || 0 == method.getSecurityMethodCoded().length()
              || 0 == method.getOption().length()) {
3             // old style segments
4             try {
5                 int code = Integer.parseInt(function.getValue());
6                 if (1 == code || 4 == code || 0 == code || 112 == code || 124
                       == code || 130 == code) {
7                     to.setProfileIdentifier(new de.xcom.tristan.bo.simple.
                          Code("RDH-1"));
8                     to.setSignatureMethod(SignatureMethod.RSA);
9                     to.setSignaturePadding(SignaturePadding.ISO9796_1);
10                    to.setSignatureHashing(SignatureHashing.RIPEMD160);
11                    to.setCipherMethod(CipherMethod.TWOKEY3DES);
12                    to.setCipherPadding(CipherPadding.CBC_ZERO_PAD);
13                    to.setKeyLength(KeyLength._700_768);
14                } else if (2 == code) {
15                    to.setProfileIdentifier(new de.xcom.tristan.bo.simple.
                          Code("DDV-1"));
16                    to.setSignatureMethod(SignatureMethod.DDV);
17                    to.setSignaturePadding(SignaturePadding.RETAILMAC);
18                    to.setSignatureHashing(SignatureHashing.RIPEMD160);
19                    to.setCipherMethod(CipherMethod.TWOKEY3DES);
20                    to.setCipherPadding(CipherPadding.CBC_ZERO_PAD);
21                    to.setKeyLength(KeyLength._128);
22                } else if (998 == code || 999 == code) {
23                    to.setProfileIdentifier(new de.xcom.tristan.bo.simple.
                          Code("PIN-1"));
24                    to.setSignatureMethod(SignatureMethod.PIN);
25                    to.setPinTanMechanism(PinTanMechanism.CLASSIC_PINTAN);
26                } else if (900 <= code && 997 >= code) {
27                    to.setProfileIdentifier(new de.xcom.tristan.bo.simple.
                          Code("PIN-2"));
28                    to.setSignatureMethod(SignatureMethod.PIN);
29                    to.setPinTanMechanism(PinTanMechanism.CHALLENGE_RESPONSE)
                          ;
30                }
31            } catch (NumberFormatException nfe) {
32                // catched to prevent abort
33            }
34        } else {
35            if (method.getSecurityMethodCoded().is("PIN")) {
36                to.setSignatureMethod(SignatureMethod.PIN);
37                if (method.getOption().is("1")) {
38            ...
```

Abbildung 4.4: Methode aus der Klasse EdiTool

Wird in der Testklasse für jede einzelne Prüfung eine separate Methode implementiert, so sind mindestens sechs Testmethoden notwendig um das angesprochene Konstrukt zu testen. Um nun die komplette Methode zu testen, ist eine hohe Anzahl von Testmethoden notwendig. Man könnte an dieser Stelle mehrere Testklassen für diese **eine** Methode implementieren um einen gewissen Überblick zu behalten.

Die Klasse besteht insgesamt aus 1.938 LOC's. Für eine gesamte Testabdeckung ist eine enorme Menge von Tests erforderlich, da alle Methoden gleichen Aufbau zeigen, wie in Abbildung 4.4 dargestellt. Es ist also mit sehr hohem Aufwand verbunden, diese eine Klasse mit einer vollständigen Testabdeckung zu versehen.

4.3.1.4 Zugriffe auf Parameter

Werden Parameter dazu genutzt weitere Instanzen zu liefern, so werden unnötige Abhängigkeiten hinzugefügt. Diese Vorgehensweise ist in einigen der genannten Klassen aus der Umfrage zu erkennen.

Die Abbildung 4.5 zeigt hierfür ein simples Beispiel. Die Deklarationen der Methode *process* setzt einen Parameter des Typs FintsProcessorContext voraus.

Aus diesem Objekt werden weitere Objekte extrahiert.

```
1     public void process(final Object order, final Object response, final
          FinTSProcessContext context) {
2         setUp(context, (Response) response);
3         process((SearchAccount1Req) order);
4     }
5
6     void setUp(final FinTSProcessContext context, final Response response) {
7         user = context.getXmlInfo().getUser();
8         client = context.getClientName();
9         acctDAO = new AcctDAO(context.getStatementProcessor());
10        adminsDAO = new AdminsDAO(context.getStatementProcessor());
11        responseAccess = new ResponseAccess(response);
12    }
```

Abbildung 4.5: Beispiel für Parameterzugriffe

Der Parameter FintsProcessorContext wird allein dazu benutzt, um weitere Objekte zur Verfügung stellen zu können. So wird in Zeile 7 mit *context.getXmlInfo().getUser()* nach einem weiteren Objekt gefragt.

Dies kann durchaus testkritisch sein, da Parameter bereits als Mockobjekte von einem Unittest bereitgestellt werden können. Von diesem Mockobjekt müssen nun weitere

Mockobjekte geliefert werden. Je länger diese Kette ist, desto komplizierter werden Tests und desto schlechter lassen diese sich warten.

Folgender Test soll dies verdeutlichen.

```
1     public void m(final D d) {
2         d.getA().getB().bf();
3     }
4
5     @Test
6     public void testm() {
7         D mockOfD = mock(D.class);
8         B mockOfB = mock(B.class);
9         A mockOfA = mock(A.class);
10
11        when(mockOfD.getA()).thenReturn(mockOfA);
12        when(mockOfA.getB()).thenReturn(mockOfB);
13
14        sut.m(mockOfD);
15        verify(mockOfB, times(1)).bf();
16    }
```

Abbildung 4.6: Beispiel Method-Chaining

In der Abbildung 4.6 benötigt man für die Zeile 2 zusätzliche Mockobjekte um den Funktionsaufruf *bf()* testen zu können. So stellen die Zeilen 8 - 12 lediglich sicher, dass entsprechende Objekte zurückgeliefert werden und ein Zugriff auf die Funktion *bf()* geprüft werden kann.

Delegiert man hingegen die Aufrufe, so das nun die Klasse *D* eine Funktion *bf()* erhält (Abbildung 4.7), so lässt sich dementsprechend ein Test für diese Methode vereinfachen, wie in Zeile 5 - 11 dargestellt.

```
1     public void m(final D d) {
2         d.bf();
3     }
4
5     @Test
6     public void testm() {
7         D mockOfD = mock(D.class);
8
9         sut.m(mockOfD);
10        verify(mockOfD, times(1)).bf();
11    }
```

Abbildung 4.7: Delegation der Informationen

4.3.1.5 Doppelter Quellcode

Ein weiteres Kriterium, dass die Testbarkeit einer Klasse beeinträchtigt, ist doppelter Quellcode. Dieser entsteht meist durch das Kopieren von Quellcode. Doppelter Quellcode führt dazu, dass auch Tests mehrmals implementiert werden müssen. Ebenso sind bei Änderungen immer mehrere Stellen betroffen. Dies wurde von mehreren Entwicklern genannt, konkrete Beispiele konnten jedoch nicht aufgezeigt werden.

Damit auch doppelter Quellcode in das Qualitätsmodell aufgenommen werden kann, soll im nächsten Abschnitt ein Refactoring durchgeführt werden. Dieses Refactoring soll sich dabei nur auf doppelten Quellcode stützen. Ziel ist es dabei, Indizien zu finden, die Entwickler bereits angesprochen hatten. Auch soll dadurch die Einordnung zu einem Merkmal offensichtlicher werden.

4.3.1.6 Kohäsion

Die Kohäsion einer Klasse beschreibt den Zusammenhalt der Daten, mit der ein Objekt arbeitet. Eine hohe Kohäsion bedeutet, dass eine Klasse eine gute Abstraktion bildet und mit einer wohldefinierten Aufgabe vertraut ist.

Wird dieses Prinzip verletzt, so erledigt eine Klasse zu viele Aufgaben. Dies hat zur Folge, das mehr Tests zu dieser Klasse implementiert werden müssen als notwendig. Dadurch steigt der Umfang der Testklasse.

Die Kohäsion kann ebenso mit dem Single-Responsibility-Principle (SRP) beschrieben werden. Robert C. Martin [Mar05, Seite 246] definiert SRP mit: “Every Class should have a single responsibility: It should have a single purpose in the system, and there should be only one reason to change.“ Ebenso wird von Robert C. Martin [Mar02, Seite 138] SRP weiter definiert mit: ”This Principle gives us both a definition of responsibility, and a guidlines for class size.“

Berücksichtigt man diese Definitionen, so lassen sich Verantwortlichkeit und der Umfang einer Klasse einschränken.

Da das Wissen um Softwaremetriken in der Abteilung nicht sonderlich ausgeprägt ist, werden solche Tatsachen von den Entwicklern nicht berücksichtigt. Ebenso wenig wurde dies in der Umfrage von den Entwicklern so dargestellt. Dennoch soll im nächsten Abschnitt SRP näher untersucht und mit Metriken bewertet werden.

4.3.1.7 Das Open-Closed-Principle (OCP)

Robert C. Martin [Mar05, Seite 287] definiert OCP mit: “The idea behind is that code should be open for extension but closed to modification. What does this mean? It means that when we have good design, we just don’t have to change code much to add new features.“

Ebenso sagt Robert C. Martin [Mar05, Seite 287] : ”When we remove duplication, our code often naturally starts to fall in line with the Open-Closed-Principle.“

OCP besagt, wenn neue Funktionalitäten einer Klasse hinzugefügt werden, bestehende Klassen nicht bearbeitet werden, sondern (als Beispiel) neue Klassen hinzugefügt werden müssen. Ebenso sagt Robert C. Martin, wenn OCP berücksichtigt wird, kann doppelter Quellcode minimiert werden.

Aus der Umfrage konnten keine Hinweise zu einem Verstoß von OCP gefunden werden. Dennoch soll im nächsten Abschnitt OCP untersucht und mit einer oder mehreren Metriken bewertet werden.

4.3.1.8 Ergebnis der Umfrage

Die Umfrage stellte sich als sehr schwierig heraus. Es wurden teilweise Klassen gezeigt, die in Umfang und Komplexität kaum zu übertreffen sind. An dieser Stelle ist auch den Entwicklern klar gewesen, dass sich eine solche Klasse schlecht testen lässt. Des Weiteren wurden die Informationen oft aus Sonar extrahiert. Besonders die Anzeige ”Hotspots by Complexity“ war dafür sehr beliebt.

Es gab jedoch vereinzelt auch gute Hinweise mit denen weitergearbeitet werden konnte.

Zusammengefasst kann gesagt werden, dass

- Initialisierungen von Klassenvariablen
- Initialisierungsblöcke
- Anzahl der Bedingungen
- Parameterzugriffe
- und Doppelter Quellcode

aus den Umfragen entstanden sind.

- Das Single-Responsibility-Principle (SRP)

- und Open-Closed-Principle (OCP)

haben literarischen Ursprung.

4.4 Definition der Metriken

Im folgendem Abschnitt werden die Ergebnisse aus der Entwicklerumfrage analysiert und mit Metriken bewertet. Ebenso werden existierende Metriken zur Testbarkeit vorgestellt.

4.4.1 Testbarkeit von Stefan Jungmayr

Zunächst soll untersucht werden, ob bereits existierende Ansätze im Bereich Testbarkeit genutzt werden können.

"Testability Measurement and Software Dependencies" [Jun02] von Stefan Jungmayr geht diesem Problem nach und versucht die Testbarkeit einer Klasse mit Metriken auszudrücken. Er versucht anhand von Reduktionsmetriken Stellen aufzuzeigen die testkritisch sind. Diese Metriken versuchen zu beschreiben, wie sich die Testbarkeit einer Klasse ändert, wenn bestehende Abhängigkeiten entfernt werden.

Hierzu wird beispielhaft ein bestehendes Projekt herangezogen und mit den vorgestellten Metriken von Stefan Jungmayr bewertet. Nachhaltig werden eventuell aufgedeckte testkritische Abhängigkeiten durch ein Refactoring entfernt bzw. diskutiert. Anschließend soll die Testbarkeit der Klasse mit den folgend vorgestellten Metriken bewertet werden.

4.4.1.1 Metriken zur Beurteilung der Abhängigkeitsstruktur

Class Dependency - (CD) Die Metrik drückt die direkten und indirekten Abhängigkeiten einer Klasse aus. Diese dient als Ausgangspunkt der folgenden Metriken.

Average Class Dependency - (ACD) Die Metrik gibt die durchschnittliche Klassenabhängigkeit an. Sie errechnet sich aus der Summe der CD-Metrik, geteilt durch die Anzahl der Klassen im System. ACD ist das arithmetische Mittel aus der Metrik CD.

Reduktionsmetrik Average Class Dependency - (rACD) Diese Metrik errechnet sich aus der ACD-Metrik, mit dem Unterschied, dass jeweils eine Abhängigkeit aus dem System entfernt und anschließend der ACD Wert des Systems neu berechnet wird. Die Differenz aus der ACD-Metrik mit allen Abhängigkeiten und der entfernten Abhängigkeit ergibt eine Differenz des ACD-Wertes. Je niedriger der neu errechnete ACD-Wert ist (bzw. je höher die Differenz), desto testkritischer ist diese temporär aus dem System entfernte Abhängigkeit.

Fallbeispiel XCOM-Sonar-Plugin

Folgend soll die Metrik rACD an dem Projekt **XCOM-Sonar-Plugin**[2] untersucht und bewertet werden. Ist die rACD-Metrik aussagekräftig genug, testkritische Abhängigkeiten zu erkennen, so soll diese in das Qualitätsmodell integriert werden. Andernfalls soll die rACD-Metrik keine Verwendung finden.

Das Klassendiagramm aus Abbildung 4.8 zeigt die Abhängigkeiten der einzelnen Klassen zueinander. Das Projekt besteht insgesamt aus 24 Klassen mit einem ACD-Wert von 7,667. Das bedeutet, dass im Schnitt jede Klasse von weiteren acht Klassen abhängig ist. Dabei wurden nur Klassen aus dem Namespace von **de.xcom.sonar** berücksichtigt.

Klasse	**CD(k)**
CSVExportClassScope	3
CSVExportProjectScope	4
CSVExport	21
BuildBreakMarker	1
BuildBreak	21
BuildBreakerConfiguration	21
MetricsToCover	21
XCOMMetricsPlugin	21
AnalyzeMetric	21
ChannelConfiguration	21
MessageSender	21
SonarMeasures	5
FetchPastMeasure	3

Tabelle 4.1: CD-Metriken des Projektes XCOM-Sonar-Plugin

Tabelle 4.1 zeigt die CD-Metriken des Projektes XCOM-Sonar-Plugin. Der Wert für die Metrik ACD errechnet sich aus die Summe der CD-Metrik durch die Anzahl der Klassen.

[2] siehe dazu Kapitel 5

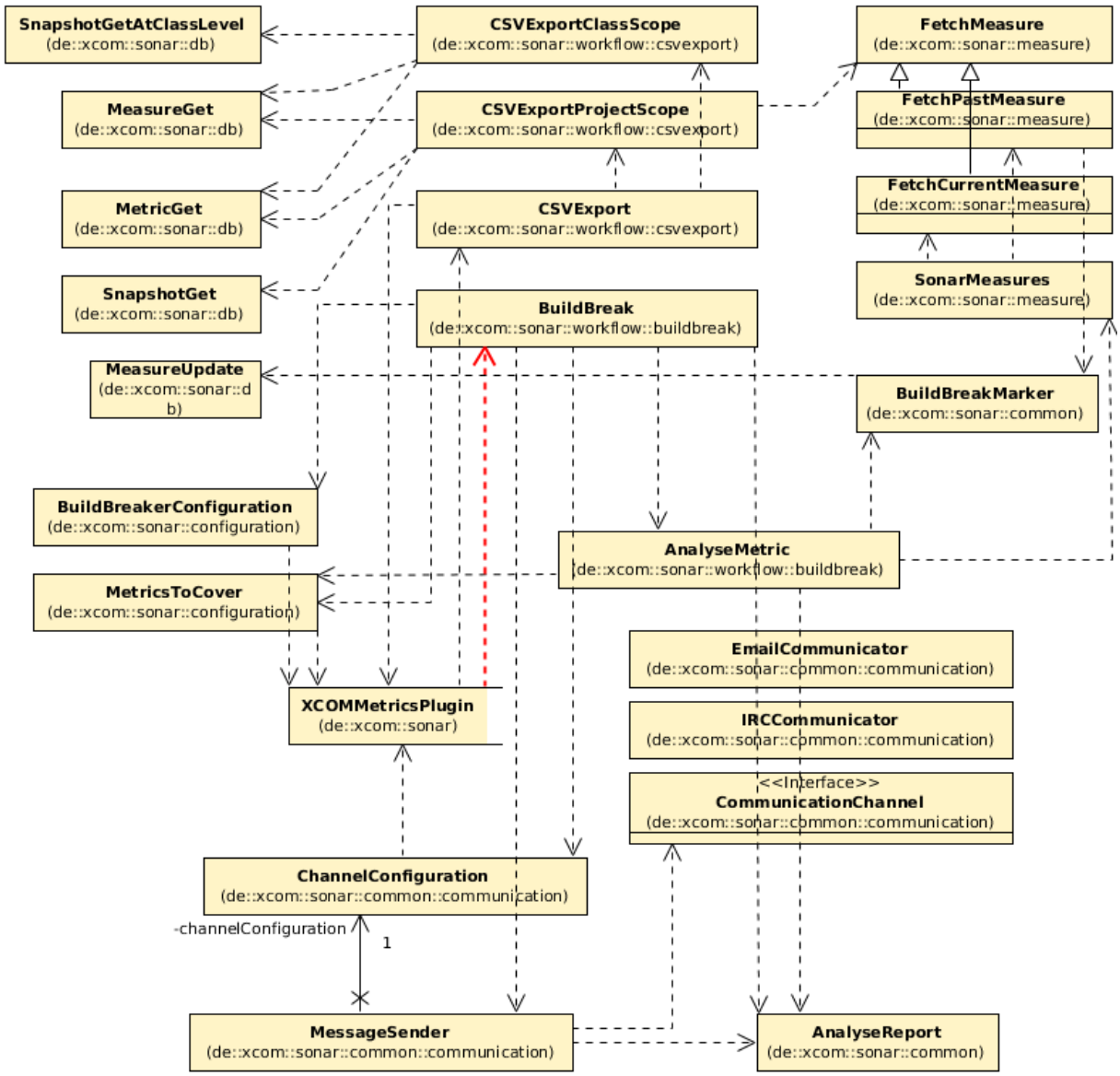

Abbildung 4.8: Klassendiagramm Sonar-Plugin

Somit ergibt $ACD = 184/24 = 7,667$. Klassen, deren Wert von $CD = 0$ ist wurden nicht in die Tabelle aufgenommen.

Für die Berechnung der Reduktionsmetrik rACD werden nun einzeln Abhängigkeiten entfernt und anschließend der ACD-Wert neu bestimmt. Die Abhängigkeit, die den größten Effekt auf die ACD-Metrik ausübt, soll anschließend näher untersucht werden.

Abhängigkeit	**ACD**	**rACD (%)**
CSVExport → CSVExportClassScope	7,0	8,7
CSVExport → CSVExportProjectScope	7,0	8,7
CSVExport → XCOMMetricsPlugin	7,083	7,62
BuildBreak → MessageSender	7,083	7,62
BuildBreak → AnalyzeMetric	5,917	22,83
BuildBreakerConfiguration → XCOMMetricsPlugin	6,792	11,41
MetricsToCover → XCOMMetricsPlugin	7,083	7,62
AnalyzeMetric → MetricsToCover	7,083	7,62
AnalyzeMetric → SonarMeasures	6,667	13,04
XCOMMetricsPlugin → BuildBreak	4,5	**41,31**
XCOMMetricsPlugin → CSVExport	5,625	26,63
ChannelConfiguration → XCOMMetricsPlugin	6,042	21,19
MessageSender → ChannelConfiguration	6,875	10,33

Tabelle 4.2: rACD-Metriken des Projektes XCOM-Sonar-Plugin

Tabelle 4.2 zeigt die Werte der Reduktionsmetriken der einzeln entfernten Abhängigkeiten. Werte von rACD < 5 wurden nicht mit in die Tabelle aufgenommen. Die Abhängigkeit XCOMMetricsPlugin → BuildBreak mit 41,31% ist laut Definition von Jungmayr die testkritischste Abhängigkeit. Diese Abhängigkeit soll nun weiter untersucht werden.

In Abbildung 4.9 ist die testkritische Abhängigkeit aufgezeigt. An dieser Stelle wird lediglich eine Liste der Erweiterungen, die Sonar zur Laufzeit ausführt, angegeben. Eine Instanz der Klasse BuildBreak wird nicht angelegt. Ebenso wenig wird auf Funktionalität der Klasse BuildBreak zugegriffen. Dies stellt somit keine testkritische Abhängigkeit dar. Jungmayr geht ebenso davon aus, dass solche testkritischen Abhängigkeiten aufgelöst werden. In diesem Beispiel würde das Sonar-Plugin nicht mehr funktionieren. Es ist also damit zu rechnen, dass die rACD-Metrik Falschaussagen liefert. Zudem lassen sich nicht immer derartige "testkritische Abhängigkeiten" entfernen, da diese durchaus einem Design-Konzept zu Grunde liegen können.

```
1     public List getExtensions() {
2         return Arrays.asList(XMetrics.class, CSVExport.class, BuildBreak.
              class);
3     }
4 }
```

Abbildung 4.9: Abhängigkeit XCOMMetricsPlugin → CSVExport

Auch wird die Abhängigkeit XCOMMetricsPlugin → CSVExport mit 26,63% als testkritisch gewertet. Hier treffen gleiche Eigenschaften wie für die Abhängigkeit XCOMMetricsPlugin → BuildBreak zu. Somit sind die zwei höchsten Werte Falschaussagen der Metrik rACD.

Die Abhängigkeit BuildBreak → AnalyzeMetric (22,83%) kann durchaus als testkritisch betrachtet werden, da dies eine fest-verdrahtete Abhängigkeit ist. Das bedeutet, dass für einen Unittest die Klasse AnalyzeMetric ebenso getestet wird, da innerhalb einer Methode AnalyzeMetric instanziiert wird.

Schlussfolgernd kann gesagt werden, dass die Art der Abhängigkeit berücksichtigt werden muss. Die Metrik rACD wird nicht ins Qualitätsmodell aufgenommen.

Class Dependency caused by Hard-Wired Dependencies - (CDh) Diese Metrik misst ebenso wie die Metrik CD die Anzahl der direkten und indirekten Abhängigkeiten auf Klassenebene. Dabei wird jedoch die Art der Abhängigkeit berücksichtigt. "Hard-Wired" beschreibt dabei die Anzahl der Abhängigkeiten, die im Fall eines Unittests mit getestet werden müssen. Die Metrik CDh gibt somit einen Wert für die Isolierbarkeit einer Klasse innerhalb eines Komponententests wieder. Je höher dieser Wert ist, umso schwieriger und komplexer können Unittests ausfallen.

Average Class Dependency caused by Hard-Wired Dependencies - (ACDh) Ähnlich der Metrik ACD, misst diese Metrik die durchschnittliche Klassenabhängigkeit von fest-verdrahteten Abhängigkeiten.

Reduktionsmetrik rACDh Die Metrik rACDh verhält sich ähnlich der rACD-Metrik. Es werden jedoch bei dieser Metrik die Abhängigkeiten aufgedeckt, die einen großen Effekt auf die durchschnittliche Anzahl der fest-verdrahteten Abhängigkeiten haben können.

Fallbeispiel XCOM-Sonar-Plugin

Folgend soll die Metrik rACDh an dem Projekt XCOM-Sonar-Plugin untersucht und bewertet werden. Ist die rACDh-Metrik aussagekräftig genug, testkritische Abhängigkeiten zu erkennen, so soll diese in das Qualitätsmodell integriert werden. Andernfalls soll die rACDh-Metrik keine Verwendung finden.

Klasse	**CDh(k)**
CSVExportClassScope	3
CSVExportProjectScope	4
CSVExport	7
BuildBreak	5
FetchPastMeasure	3
FetchCurrentMeasure	1
SonarMeasures	5
BuildBreakMarker	1
BuildBreak	12
AnalyzeMetric	7

Tabelle 4.3: CDh-Metriken des Projektes XCOM-Sonar-Plugin

Abhängigkeit	**ACDh**	**rACDh (%)**
CSVExportClassScope → SnapshotGetAtClassLevel	1,708	4,69
CSVExportProjectScope → SnapshotGet	1,708	4,69
CSVExportProjectScope → FetchMeasure	1,708	4,69
CSVExport → CSVExportClassScope	1,708	4,69
CSVExport → CSVExportProjectScope	1,667	6,98
FetchPastMeasure → BuildBreakMarker	1,625	9,32
SonarMeasure → FetchPastMeasure	1,583	**11,66**
SonarMeasure → FetchCurrentMeasure	1,667	6,98
BuildBreak → AnalyzeMetric	1,458	**18,64**
BuildBreakMarker → MeasureUpdate	1,583	**11,66**
AnalyzeMetric → SonarMeasure	1,458	**18,64**
AnalyzeMetric → AnalyzeReport	1,708	4,69

Tabelle 4.4: rACDh-Metriken des Projektes XCOM-Sonar-Plugin

Das Klassendiagramm aus Abbildung 4.10 zeigt die fest-verdrahteten Abhängigkeiten. Der Wert der Metrik ACDh beträgt 1,75. Das bedeutet, dass im Schnitt jede Klasse mit zwei weiteren Klassen fest-verdrahtet ist. Auch hier wurden nur Klassen aus dem Namespace von **de.xcom.sonar** berücksichtigt.

Wie in Tabelle 4.4 zu erkennen, wurden die Abhängigkeiten BuildBreak → AnalyzeMetric

und AnalyzeMetric → SonarMeasure mit 18,64% als die testkritischsten Abhängigkeiten eingestuft.

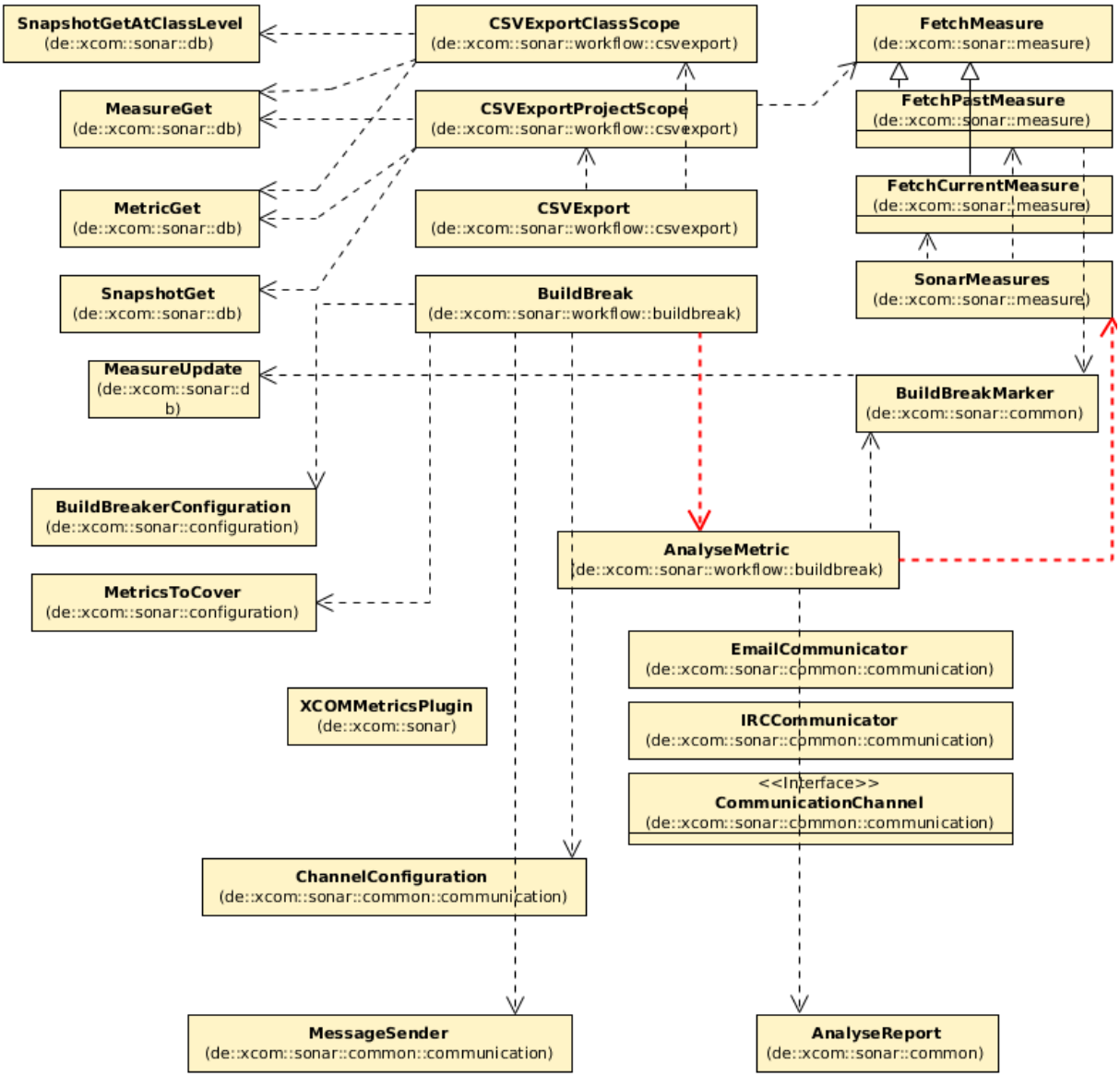

Abbildung 4.10: Klassendiagramm Sonar-Plugin mit fest-verdrahteten Abhängigkeiten

In dem Klassendiagramm 4.10 wurden diese Abhängigkeiten mit Rot gekennzeichnet. Man kann gut erkennen, dass diese Abhängigkeiten den längsten Abhängigkeitsgraph bilden.

Möchte man die Klasse BuildBreak zu 100% mit Tests abdecken, so werden automatisch die Klassen

- AnalyzeMetric
- AnalyzeReport
- BuildBreakMarker
- MeasureUpdate
- SonarMeasures
- FetchCurrentMeasure
- FetchPastMeasure
- und FetchMeasure

mit getestet. Dies kann durchaus testkritisch sein, da zum einen ein Testen nicht möglich sein kann (Datenbankverbindung, Zugriff auf Dateisystem) und zum anderen die Tests unnötig kompliziert werden bzw. ein kompliziertes Setup benötigen.

Weitere Metriken von Jungmayr sollen nicht besprochen werden. Auch bei folgenden Metriken von Stefan Jungmayr wird der Typ der Abhängigkeit nicht berücksichtigt:

- NCDC
- rNCDC
- NFD
- rNFD

Ergebnis

Die Metriken CDh, ACDh und rACDh stellen eine gute Möglichkeit dar, die Testbarkeit einer Klasse anhand ihrer Abhängigkeiten zu beschreiben. Die Metriken CDh, ACDh und rACDh werden in das Qualitätsmodell aufgenommen.

4.4.2 Anzahl der Bedingungen

Entwicklerbefragung

Die weitere Befragung der Entwickler ergab, dass Klassen mit vielen Bedingungen als aufwendiger betrachtet werden, als jene mit weniger Bedingungen, um diese mit einer hohen Testabdeckung abzusichern. Aus dieser Erkenntnis lässt sich mit Komplexitätsmetriken arbeiten, die den Programmfluss als gerichteten Graph mit Knoten und Kanten betrachtet.

Zyklomatische Komplexität nach McCabe - CCN

Thomas J. McCabe hat im Jahre 1976 die Zyklomatische Komplexität eingeführt. Er sieht Programmcode als einen gerichteten Graphen mit Knoten und Kanten. Es wird die Anzahl der Kanten mit der Anzahl der Knoten und Teilgraphen ergänzt. [Sne10, Seite 164] Man nutzt hierbei die Graphentheorie.

Die Formel zur Berechnung der Zyklomatischen Komplexität lautet: $V(s) = e - n + 2p$
Dabei sind:

- e - die Anzahl der Kanten
- n - die Anzahl der Knoten
- p - die Anzahl der unabhängigen Teilgraphen (bzw. Anzahl der Methoden)

Für die Berechnung der Komplexität einer einzelnen Methode, die nur einen Aus- und Eingang besitzt, errechnet sich die McCabe-Metrik mit $V(s) = 1 + e - n$.

Festgelegt wurde von McCabe, dass die Zyklomatische Komplexität eines Codebausteins den Wert 10 nicht überschreiten darf. Dies kann auf jede Methode einer Klasse transformiert werden. Um Aussagen über die Komplexität einer Klasse machen zu können, muss die Komplexität einer jeden Methode betrachtet werden.

Beispiel

In Abbildung 4.11 ist eine Methode mit CCN = 7 dargestellt.

```
1     public Measure get(final Metric metric) {
2         List<Measure> measures = getAll(metric);
3         if (measures.isEmpty()) {
4             return new Measure(metric, 0D);
5         }
6         if (onlyCurrentMeasureAvailable(measures)) {
7             return measures.get(0);
8         }
9         BuildBreakMarker buildBreaker = new BuildBreakMarker();
10        if (null != buildBreaker.getMeasureWithBuildBreakEntry(measures)) {
11            return buildBreaker.getMeasureWithBuildBreakEntry(measures);
12        }
13        return measures.get(measures.size() - 2);
14    }
```

Abbildung 4.11: CCN-Beispiel

Um den Wert für diese Methode zu berechnen, reicht ein simples Zählen von if, for, while, case, catch, throw, return(das nicht die letzte Anweisung in einer Methode ist), &&, || und ?. Dabei besitzt jede Methode den Grundwert 1.

Der Fall, die Komplexität der Klasse mit Hilfe der McCabe-Metrik zu beschreiben, soll nicht geschehen. Die McCabe Metrik hilft in diesem Fall, einen Testaufwand zu beschreiben.

Feststellung benötigter Testfälle

Folgend soll untersucht werden, inwieweit die Komplexität mit der Anzahl von Testfällen ausgedrückt werden kann. Das Beispiel aus Abbildung 4.11 benötigt für eine 100%-ige Testabdeckung insgesamt vier Testfälle. Die Sinnhaftigkeit der implementierten Testfälle soll dabei nicht untersucht werden.

Für einen kompletten Testdurchlauf muss es eine Anzahl von Messungen geben (Zeile 2). Ebenso müssen mindestens zwei Messungen zur Verfügung stehen (Zeile 6). Die Messungen dürfen keine Abbruchbedingungen enthalten(Zeile 10). Sind diese Anforderungen im Test erfüllt, so wird mit einem Test bereits die Komplexität mit einem Wert von vier abgearbeitet. Weitere drei Tests sind nötig, um die enthaltenen Bedingungen zu prüfen. Daraus wird ersichtlich, dass die Anzahl von Tests für eine komplette Testabdeckung nicht direkt von der Komplexitätszahl abgelesen werden kann.

Man kann jedoch explizit davon ausgehen, wenn aus dieser Methode Bedingungen entfernt bzw. hinzugefügt werden, dass auch der Testaufwand für diese Methode **und** der zugehörigen Klasse beeinträchtigt wird.

Das folgende Beispiel soll dies weiter verdeutlichen. Es soll mit Hilfe der Darstellung eines Datenflusses die Anzahl der notwendigen Testfälle ermittelt werden.

```
1     if (null != data && 5 <= data.length) {
2         //doSomething
3     }
```

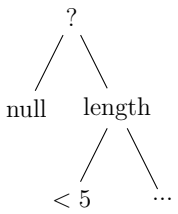

Abbildung 4.12: Datenfluss als Binärbaum

Das Beispiel aus Abbildung 4.12 zeigt die Kanten der entsprechenden Bedingung, die für ein maximales Testcoverage durch Unittests getestet werden müssen. Die Komplexität der Bedingung beträgt 2 (Anzahl Kanten - Anzahl Knoten), wobei jedoch 3 Testfälle (Anzahl der Blätter eines Binärbaums) notwendig sind.

```
1     if (null != data && 5 <= data.length && 20 => data.length)) {
2         //doSomething
3     }
```

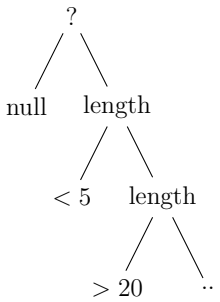

Abbildung 4.13: Datenfluss als Binärbaum mit erhöhter Komplexität

Wird nun eine weitere Bedingung (oder ein weiterer Knoten) hinzugefügt, so erhöht sich auch der damit verbundene Testaufwand. Nun müssen 6 Branches abgesichert werden,

wofür 4 Testfälle notwendig sind. Die Anzahl der Blätter eines Binärbaums entspricht der Anzahl der notwendigen Testfälle für ein maximales Testcoverage. Da jedoch auch ein Testaufwand mit Hilfe der Komplexitätsmetrik beschrieben werden kann, soll vorerst diese Metrik Verwendung finden.

Es lässt sich nicht die Anzahl der Testfälle mit der Komplexitätsmetrik ermitteln. Jedoch kann festgehalten werden, dass auch der Testaufwand steigt wenn die Komplexität einer Klasse zunimmt. So muss ebenfalls damit gerechnet werden, dass bei hohen Komplexitätswerten ebenso eine hohe Anzahl von Testfällen notwendig wird.

False-Positive

Falschaussagen zwecks der Komplexität sind dennoch möglich. Werden Methoden weiter refaktorisiert (beispielsweise das Extrahieren neuer Methoden), so erhöht sich die Komplexität der Klasse, da auch private Methoden mit einem Grundwert betrachtet werden. An dieser Stelle werden jedoch keine neuen Knoten hinzugefügt und somit müssen auch keine neuen Testfälle erstellt werden.

Entwickler würden eher vom einem Refactoring absehen, wenn sich die Komplexität der Klasse dabei erhöht und die Wertung des Testaufwandes negativ beeinträchtigt wird.

Grundsätzlich ist es also besser, nur öffentlichen Methoden einen Grundwert zu geben. Private Methoden sollten die Komplexität einer Klasse nicht erhöhen wenn der Testaufwand mittels der Komplexitätszahl ausgedrückt werden soll.

Verfügbarkeit in Sonar

Die CCN-Metrik wird von Sonar zur Verfügung gestellt. Hier kann auf die Klassenkomplexität zurückgegriffen werden. Die Klassenkomplexität errechnet sich aus der Summe der Komplexitäten jeder Methode. Die Methodenkomplexität beruht auf der McCabe-Metrik. Mit der zusätzlichen Anforderung, private Methoden nicht mit einem Grundwert zu bewerten, muss eine neue Metrik definiert werden. Diese kann jedoch auf der Klassenkomplexitätsmetrik aufbauen. Zusätzlich wird die Anzahl der Methoden und die Anzahl der öffentlichen Methoden einer Klasse benötigt. Diese werden von Sonar zur Verfügung gestellt.

Ergebnis

Die Klassenkomplexität ist ein guter Anhaltspunkt, den Testaufwand in Zahlen auszudrücken. Private Methoden sollen jedoch keinen Grundwert bekommen.

4.4.3 Doppelter Quellcode

Entwicklerbefragung

Bei der Befragung der Entwickler wurden keine konkreten Beispiele gezeigt. Es wurde jedoch gesagt, dass doppelter Quellcode durchaus ein Testen erschweren bzw. den Testaufwand steigern kann. Hierzu wurde ein Refactoring, um dies bestätigen zu können, durchgeführt.

Bei diesem Refactoring wurde ausschließlich mit der Metrik Duplicated Blocks aus Sonar gearbeitet.

Refactoring von Tristan - End of Day processing

Das Projekt "Tristan - End of Day processing" wurde für das Refactoring ausgewählt.

Tristan - EOD	Vor dem Refactoring	Nach dem Refactoring
Duplicated Blocks	221	4
Klassen	456	452
Lines of Code	14501	13328
Complexity/Class	4,7	4,6
LCOM4 [a]	1,1	1,0

Tabelle 4.5: Metriken von Tristan EOD

[a] siehe 4.4.6 Metrik Lack of Cohesion of Methods (LCOM)

Das Ziel war es, nur doppelten Quellcode zu entfernen ohne andere Metriken näher in Sonar zu betrachten. Wie in Tabelle 4.5 zu sehen, hat sich der Umfang des Projektes um knapp 1200 Codezeilen verringert. Dies lässt vermuten, dass ebenso Tests für die entfallenen Codezeilen überflüssig sind. Somit wäre der Aufwand in der Entwicklungsphase geringer gewesen, wenn keine oder nur wenige Duplicated Blocks entstanden wären.

In Abbildung 4.14 ist die Methode *process* aus der Klasse UserProcessor vor dem Refactoring dargestellt. Im Anhang A.1.3 befindet sich zusätzliches Material.

Diese Klasse zeigt doppelten Quellcode, der über das Webinterface von Sonar dargestellt werden kann.

Die Abbildung 4.15 macht dies deutlich. So sind der *Catch-Block*, die Methode *checkDeleteMethod* und *handleException* identisch mit denen aus der Klasse AdminsProcessor. Auch übernimmt jeder Prozessor einen Teil des Transaction-Handlings.

```
1     @Override
2     public EodProtocol process(final EodConfiguration eodConfig) throws
          EodException {
3         DeleteMethod deleteMethod = new DeleteMethod(eodConfig.
              getDeleteMethod());
4         checkDeleteMethod(deleteMethod);
5         EodProtocol eodProtocol = new EodProtocol();
6         statementProcessor.turnOffAutoCommit();
7         try {
8             String userStoragePeriod = eodConfig.getStoragePeriod();
9             Collection<User> users = getUsersToDelete(Integer.valueOf(
                  userStoragePeriod));
10            if (isEmpty(users)) {
11                log.info("No Users found to delete...");
12                eodProtocol.setInfo("No Users found to delete...");
13            } else {
14                log.info("Found " + users.size() + " User(s) to delete");
15                log.info("Deleting records older than " + userStoragePeriod +
                       " day(s) now...");
16                processUsers(users, deleteMethod);
17                eodProtocol.setInfo(users.size() + " User(s) deleted");
18                eodProtocol.setDeleted(users.size());
19            }
20            statementProcessor.turnOnAutoCommit();
21            log.info("done ...");
22        } catch (SQLException e) {
23            handleException(e);
24        } catch (EodQueryException e) {
25            handleException(e);
26        }
27        return eodProtocol;
28    }
```

Abbildung 4.14: Klasse UserProcessor vor dem Refactoring

Diese Aufgaben können eine Abstraktionsebene höher ausgelagert werden. Da es insgesamt 13 Prozessoren gibt, die ähnliche Aufgaben übernehmen, entfallen Tests für die entsprechenden Funktionen.

Diese Tests können dann einmalig in der übergelagerten Klasse implementiert werden. Damit wird der Testaufwand in den einzelnen Prozessoren eingeschränkt.

In Abbildung 4.16 ist die Methode *process* refaktorisiert wurden. Ebenso wurde die Methode *checkDeleteMethod* eine Abstraktionsebene höher ausgelagert. Um das Exception-Handling sowie das Transaction-Handling muss sich die Klasse UserProcessor ebenfalls nicht mehr kümmern. Dies wurde auch in allen anderen Prozessoren so umgesetzt.

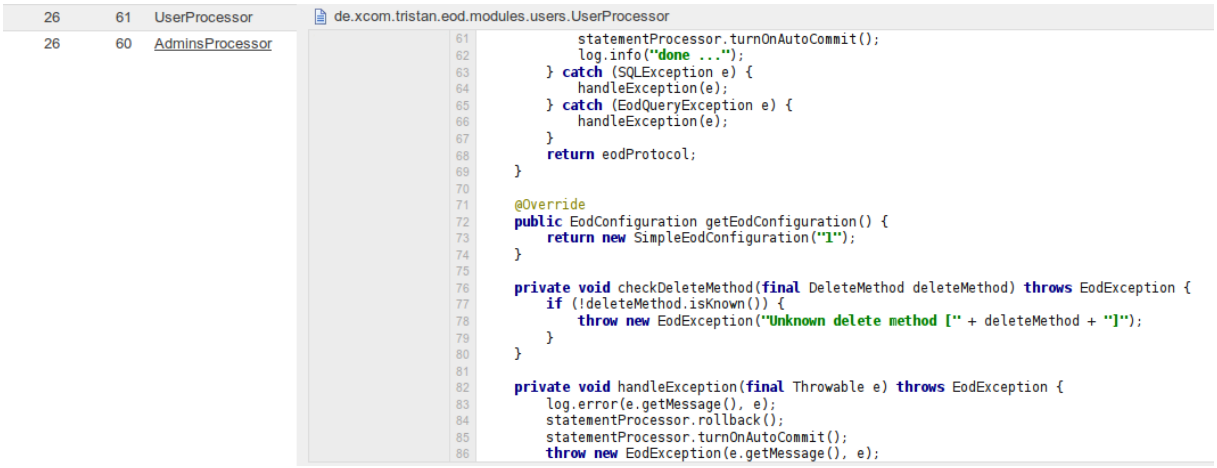

Abbildung 4.15: Duplicationsexplorer von Sonar

```
1   @Override
2   public EodProtocol process(final EodConfiguration eodConfig) throws
        Exception {
3       DeleteMethod deleteMethod = new DeleteMethod(eodConfig.
            getDeleteMethod());
4       EodProtocol eodProtocol = new EodProtocol();
5       String userStoragePeriod = eodConfig.getStoragePeriod();
6       Collection<User> users = getUsersToDelete(Integer.valueOf(
            userStoragePeriod));
7       if (isEmpty(users)) {
8           log.info("No Users found to delete...");
9           eodProtocol.setInfo("No Users found to delete...");
10      } else {
11          log.info("Found " + users.size() + " User(s) to delete");
12          log.info("Deleting records older than " + userStoragePeriod + "
                day(s) now...");
13          processUsers(users, deleteMethod);
14          eodProtocol.setInfo(users.size() + " User(s) deleted");
15          eodProtocol.setDeleted(users.size());
16      }
17      log.info("done ...");
18      return eodProtocol;
19  }
```

Abbildung 4.16: Klasse UserProcessor nach dem Refactoring

Zusammengefasst kann gesagt werden, dass weniger Tests der einzelnen Klassen notwendig sind um die gesamte Klasse zu testen. Ebenso wurde die Komplexität der Tests verringert, da Exception- und Transaction-Handling ausgelagert wurde. Auch lässt sich aus Abbildung 4.15 ablesen, dass das Exception- und Transaction-Handling ”zusätzliche“ Komplexitäten enthält.

Die Metrik Duplicated Blocks

Das Refactoring wurde mit Hilfe der Metrik Duplicated Blocks durchgeführt. Sonar stellt diese Metrik über das Plugin PMD[3] bereit. PMD beinhaltet einen Copy/Paste Detector (CPD) und unterstützt standardmäßig Java, JSP, C, C++, Fortran und PHP. Sonar bietet ebenfalls eine proprietäre CPD-Engine an.

Duplicated Blocks baut auf Duplicated Lines auf. Duplicated Lines werden ab einer definierten Anzahl von Zeichen[4] aufgenommen. Wenn sich als Beispiel 30 Zeilen aus 2 Dateien oder Klassen gleichen, dann gilt dies als ein Duplicated Block.

Gibt es keine Duplicated Lines, so gibt es auch keine Duplicated Blocks.

Verfügbarkeit in Sonar

Die Metrik Duplicated Blocks wird in Sonar zur Verfügung gestellt. Um einen zusätzlichen Testaufwand beschreiben zu können, sollte jedoch die Komplexität von Duplicated Blocks betrachtet werden. Dies wird von Sonar nicht zur Verfügung gestellt.

Ergebnis

Die Metrik Duplicated Blocks gibt gute Anhaltspunkte, einen zusätzlichen Testaufwand in Zahlen ausdrücken zu können. Dafür soll die Komplexität eines Duplicated Blocks betrachtet werden.

4.4.4 Fest-verdrahtete Abhängigkeiten

Entwicklerbefragung

Aus der Entwicklerbefragung ist bekannt, das ein Initialisieren von Klassenvariablen innerhalb eines Konstruktors schwer zu testen ist. Geht man dieser Tatsache etwas genauer auf den Grund, so sind fest-verdrahtete Abhängigkeiten das Problem. Gleiches Problem besteht bei Implementierungen von Initialisierungsblöcken. So ist die Benutzung des new-Operators innerhalb eines Konstruktors oder einer Methode testkritisch.

[3] siehe dazu http://pmd.sourceforge.net/
[4] Standard: ab 120 Zeichen werden doppelte Quellcodezeilen aufgenommen

Die Metriken CDh, ACDh und rACDh

Im Abschnitt 4.4.1 Testbarkeit von Stefan Jungmayr wurden diese Metriken bereits untersucht. Diese stellen mit der Anzahl der Abhängigkeiten, die nicht durch ein Mockobjekt ersetzt werden können, einen Wert bereit, mit der eine Klasse auf Isolierbarkeit bewertet werden kann. Ebenso sind besonders auffällige Abhängigkeiten ein guter Indikator, ein Refactoring an den entsprechenden Stellen durchzuführen.

Verfügbarkeit in Sonar

Die Metriken CDh, ACDh und rACDh werden von Sonar nicht unterstützt. Eine Untersuchung des Ticketsystems von Sonar[5] ergab keine Treffer auf eventuelle zukünftige Implementierungen. Ebenso erfolgte eine Anfrage an Stefan Jungmayr, das Metrik-Tool ImproveT für wissenschaftliche Zwecke zur Verfügung zu stellen. Herr Jungmayr hat jedoch angesprochen, dass das Tool ImproveT ausschließlich als Plugin konzipiert wurde und in der Entwicklungsumgebung Together eingesetzt wird. Ebenfalls wurde gesagt, dass an dem Tool ImproveT seit geraumer Zeit nicht mehr gearbeitet wurde. Die Verwendung von ImproveT ist innerhalb von Sonar nicht möglich.

Ergebnis

Um Aussagen über testkritische Abhängigkeiten treffen zu können, müssen die Metriken CDh, ACDh und rACDh in Sonar implementiert werden.

4.4.5 Zugriffe auf Parameter

Entwicklerbefragung

Aus der Entwicklerbefragung geht hervor, dass bestimmte Zugriffe auf Parameter als testkritisch betrachtet werden können.

Law of Demeter

Robert C. Martin [Mar09, Seite 97] sagt: „A module shout not known about the innards of the *objects* its manipulates.“ Diese Regel definiert er als Law of Demeter (LOD).

[5] siehe dazu `http://jira.codehaus.org`

Unter [RW12] wird beschrieben, ein Zusammenspiel der Objekte zu beschränken. Dies wird ebenfalls als LOD definiert und besagt, dass eine Methode nur folgende andere Methoden aufrufen darf:

1. Methoden der eigenen Klasse
2. Methoden der Parameter einer Methode
3. Methoden assoziierter Klassen
4. Methoden selbst erzeugter Objekte innerhalb einer Methode

Ebenso wird gesagt, dass Klassen die als reine Datenhaltungsklassen dienen (Entity-Klassen), von LOD ausgeschlossen werden können.

LOD beschreibt dabei den Zugriff auf Objekte über Methoden und berücksichtigt dabei nicht nur die Parameter einer Methode, ebenso werden Klassenvariablen und selbsterzeugte Objekte einer Methode untersucht. Dies konnte so aus der Umfrage nicht ermittelt werden, soll jedoch berücksichtigt werden, da es gleiches Verhalten in einem Test hervorrufen kann.

Metrik VOD - Violation of Demeter

Eine Metrik für LOD wird in [Ojh94, Seite 34] erwähnt. Diese wird mit Violation of Demeter (VOD) beschrieben. Definiert wird VOD mit: “The Number of messages sent to classes which are not parameters to messages or explicitly declared within the class.” Ebenso wird gesagt, dass die Metrik VOD die Prinzipien von LOD befolgt.

Metric VOD - Beispiele für Verstöße

Beispiel 1: Methoden der Parameter einer Methode und assoziierter Klassen

```
1 public class Foo {
2
3     private Bar bar;
4
5     private void doSomething(final Param param) {
6         param.doSomething(); // keine Verletzung von LoD
7         param.getObject().doSomething(); //Verletzung von LoD
8         param.getObject().getAnotherObject().doSomething(); //Verletzung von LoD
9         param.getBar().doSomething(); // keine Verletzung von LoD
10    }
11 }
```

Abbildung 4.17: LOD Beispiel 1

Die Zeilen 7-8 zeigen einen Verstoß von LOD. An diesen Stellen wird über das Objekt *param* nach weiteren Objekten gefragt, die der Klasse *Foo* nicht bekannt sind.

In Zeile 9 wird über das Objekt *param* nach einem Objekt *Bar* gefragt. Dies stellt keine Verletzung von LOD dar, da die Klasse *Bar* der Klasse *Foo* bereits bekannt ist.

Beispiel 2: Methoden selbst erzeugter Objekte innerhalb einer Methode

```
1 public class Foo {
2
3     private Bar bar;
4
5     private void doSomething(final Param param) {
6         Object object = new Object();
7         object.doSomething(); // keine Verletzung von LoD
8         param.doSomething(); // keine Verletzung von LoD
9         param.getObject().doSomething(); // keine Verletzung von LoD
10        param.getObject().getAnotherObject().doSomething(); //Verletzung von LoD
11        param.getBar().doSomething(); // keine Verletzung von LoD
12    }
13 }
```

Abbildung 4.18: LOD Beispiel 2

Zeile 9 stellt nun keinen Verstoß von LOD dar, da in Zeile 6 bereits gleicher Typ assoziiert wird.

Beispiel 3: Abstrakte Darstellungsweise

Folgendes Beispiel soll die Sicht aus einer abstrakteren Sichtweise darstellen.

Wenn man eine Ware kaufen möchte und diese an einer Kasse bezahlt, so wird man seinen Geldbeutel öffnen, einen bestimmten Betrag aus dem Geldbeutel entnehmen und anschließend bezahlen. Man könnte an dieser Stelle ebenfalls seinen Geldbeutel übergeben und das Geld herausnehmen lassen. So müsste zur Verifizierung der Geldbetrag beispielsweise gezeigt werden, bevor die eingekaufte Ware als bezahlt gilt. Man kann dieses Beispiel noch weiter ausbauen. So könnte man beispielsweise der Kassiererin seine Jacke überreichen, in der sich die Geldbeutel befindet. Diese müsste den Geldbeutel aus der Jacke entnehmen und den Geldbetrag aus dem Geldbeutel entnehmen.

So lässt sich auch der Verantwortungsbereich einer Klasse einschränken. Die Implementierung der Kassiererin würde dann anstatt eines Parameters Geldbeutel oder Jacke, lediglich den Betrag fordern. Das Bereitstellen des Geldbetrages liegt in der Verantwortung des Kunden.

Verfügbarkeit in Sonar

Mit der Version 2.13.1 von Sonar wird keine Metrik für VOD unterstützt. Es existiert jedoch ein Ticket für die Metrik VOD im Sonar-Ticketsystem. Dies wurde aufgrund von mangelnder Beschreibung jedoch abgelehnt.

PMD bietet ebenfalls ab der Version 5.0[6] eine Metrik für VOD an. Diese Metrik nennt sich "LawOfDemeterRule" und wird wie folgt definiert: "This rule can detect possible violations of the Law of Demeter. The Law of Demeter is a simple rule, that says only talk to friends. It helps to reduce coupling between classes or objects."

Ergebnis

Die Metrik VOD muss implementiert werden. In Kapitel 5 Prozessintegration wird über die Umsetzung der Metrik VOD diskutiert.

4.4.6 Das Single-Responsibility-Principle (SRP)

Entwicklerbefragung

[6] benutzte PMD-Version von Sonar ist 4.3

Die Entwicklerbefragung ergab keine Anhaltspunkte, das ein Verstoß gegen SRP als testkritisch betrachtet werden muss. Berücksichtigt man die Definitionen von Robert C. Martin und das bereits durchgeführte Refactoring im Abschnitt 4.4.3 Doppelter Quellcode, so kann SRP mit der Metrik LCOM in Verbindung gebracht werden.

Refactoring von Tristan - End of Day processing

Am folgendem Beispiel soll die Metrik LCOM4 und SRP demonstriert werden. In der Abbildung 4.19 ist die Klasse CustomerGetLastDeleteId vor dem Refactoring dargestellt.

```
1  public class CustomerGetLastDeleteId implements QueryContext<StringResult> {
2
3      private Log log = LogFactory.getLog(CustomerGetLastDeleteId.class);
4      private DBComplexObject dbComplexObject = new DBComplexObject();
5
6      public CustomerGetLastDeleteId(final String client, final String
           customerId) {
7          dbComplexObject.add(new DBString(customerId));
8          dbComplexObject.add(new DBString(client));
9      }
10
11     @Override
12     public String getSQL() {
13         return "select max(CUSTOMER_ID)"
14                 + " from CUSTOMER"
15                 + " where CUSTOMER_ID like ? || '~DEL%'"
16                 + " and CLIENT = ?";
17     }
18
19     @Override
20     public void setInputParameters(final PreparedStatement statement) throws
           SQLException {
21         StatementPrepare statementPrepare = new StatementPrepare(log);
22         statementPrepare.prepareSelect(statement, getSQL(), dbComplexObject);
23     }
24
25     @Override
26     public StringResult getResult(final ResultSet resultSet) throws
           SQLException {
27         if (resultSet.next()) {
28             return new StringResult(resultSet.getString(1));
29         }
30         return new StringResult(null);
31     }
32 }
```

Abbildung 4.19: Klasse CustomerGetLastDeleteId vor dem Refactoring

Der Wert der Metrik für LCOM4 dieser Klasse beträgt zwei, wie in Abbildung 4.20 zu sehen. Diese Darstellungsweise lässt vermuten, dass diese Klasse zwei Aufgaben erledigt

und ebenfalls zwei Zugriffspfade anbietet.

Über das Interface PreparedStatement[7] wird ein SQL-Statement vorbereitet und mit entsprechenden Parametern aus dem Objekt *dbCOmplexObject* gefüllt. Das Ausführen der Abfrage wird dann von einer separaten Klasse ausgeführt.

Anschließend muss die Klasse CustomerGetLastDeleteId das Ergebnis dieser Abfrage bereitstellen. Dafür wird die Klasse StringResult mit den Daten aus dem Abfrageergebnis gefüllt.

Aus der Beschreibung der Aufgabe wird bereits klar, dass die Klasse CustomerGetLastDeleteId zwei Aufgaben zu erledigen hat, was einen Verstoß von SRP entspricht.

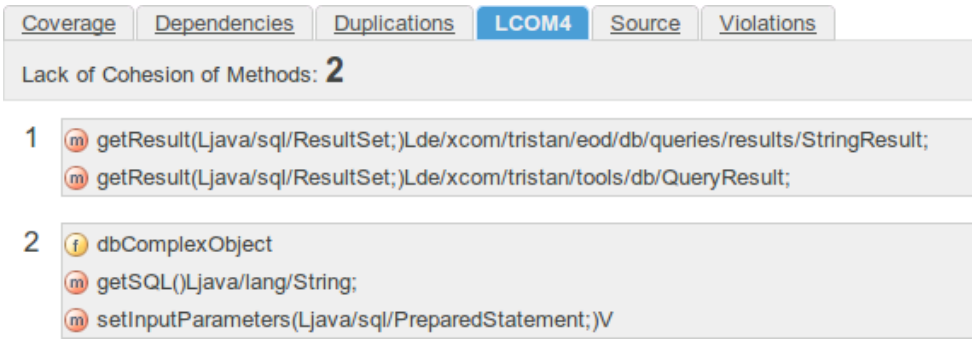

Abbildung 4.20: LCOM4 Metrik der Klasse CustomerGetLastDeleteId

Trifft man Überlegungen, das Bereitstellen des Ergebnisses an eine andere Klasse zu übertragen, entfallen mindestens zwei Tests für die Methode *getResult*. Man könnte beispielsweise die Funktionalität der Methode *getResult* zunächst vollständig in die Klasse StringResult auslagern.

Im Projekt "TRISTAN - Standard Administration Subsysteme" findet diese Klasse 67mal Verwendung. Würde an dieser Stelle dieses Refactoring durchgeführt werden, so wären ca. 130 Tests nicht mehr nötig. Auch wäre doppelter Quellcode in den Tests geringer.

Diese Refactoring kann ebenso auf weitere Klassen, die das Interface QueryResult implementieren, angewendet werden. Dadurch würden sich viele Tests erübrigen und die Klassen würden weniger Aufgaben erledigen und wären somit übersichtlicher.

[7] siehe dazu: http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/PreparedStatement.html

Metrik Lack of Cohesion of Methods (LCOM)

Die Metrik Lack of Cohesion of Methods (Fehlende Kohäsion in Methoden) existiert in 4 Versionen [8]. LCOM wurde im Jahre 1993 in der Chidamber & Kemerer Object-Oriented Metrics Suite vorgestellt. [Kem93]

Es soll jedoch ausschließlich die aktuelle Version LCOM4 betrachtet werden.

LCOM4 betrachtet Methoden und Felder als eine zusammenhängende Komponente. LCOM4 greift dabei auf die Graphentheorie zurück (siehe McCabe). Dabei sind Knoten Methoden und Felder, deren Beziehungen die Kanten sind. Entsteht mehr als eine Komponente (LCOM4 > 1), so können diese Komponenten in eine neue Klasse ausgelagert werden.

Die Abbildung 4.21 soll dies verdeutlichen.

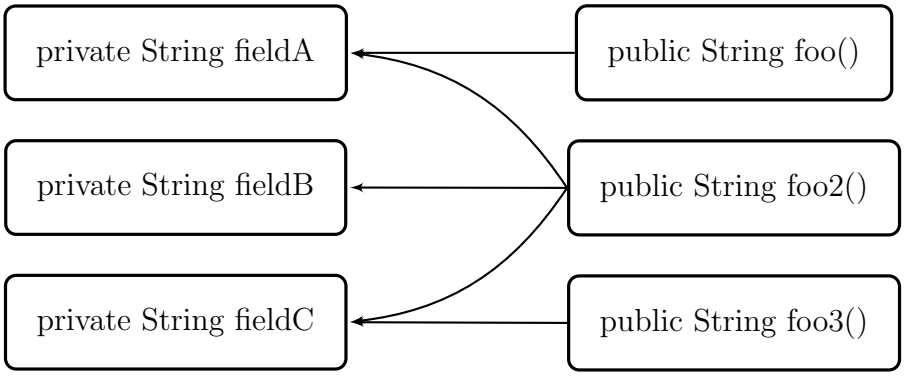

Abbildung 4.21: Beispiel für LCOM4 = 1

Hier wird LCOM4 mit 1 bewertet, da alle Methoden auf gleiche Felder zugreifen. Die Klassen und Felder bilden eine zusammenhängende Komponente.

In Abbildung 4.22 ist der Wert für LCOM4 = 2 dargestellt. Hier sind nun zwei Komponenten zu erkennen. Durch ein Refactoring könnte diese Klasse nun in eine zweite Klasse aufgeteilt werden. Dabei würde man die Methode *foo* und das Feld *fieldA* in eine neue Klasse extrahieren.

Verfügbarkeit in Sonar

Die Metrik LCOM4 steht in Sonar zur Verfügung und kann somit in das Modell integriert werden.

[8] LCOM1, LCOM2, LCOM3 und LCOM4

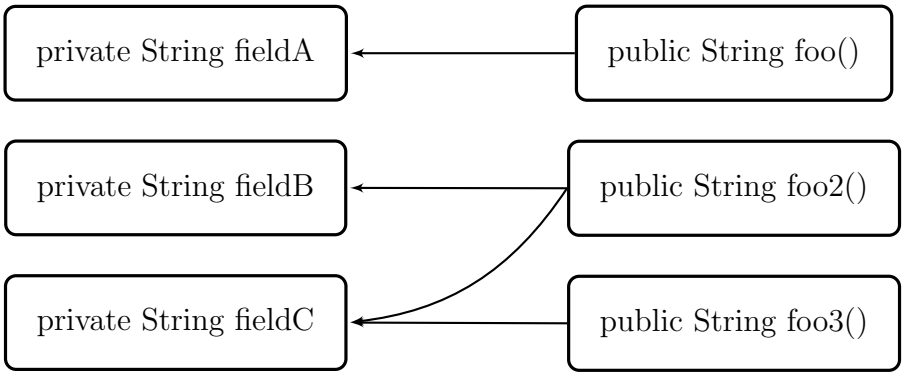

Abbildung 4.22: Beispiel für LCOM4 = 2

4.4.7 Das Open-Closed-Principle

Entwicklerbefragung

Für OCP gibt es ebenfalls keine Anhaltspunkte in der Entwicklerumfrage. Eine Betrachtung soll dennoch stattfinden, da die Definitionen von Robert C. Martin vielversprechend sind.

Beispiel eines Taschenrechners

Folgendes Beispiel soll zeigen, dass der Einsatz von Switch-Statements gegen OCP verstößt. Hierfür wurde eine kleine Anwendung implementiert, die die Funktion eines einfachen Taschenrechners übernimmt.

Abbildung 4.23 zeigt die Implementierung mit einem Switch-Statement.

```
1 public class CrazyCalculation {
2
3     private int num1;
4     private int num2;
5
6     public CrazyCalculation(final List<Integer> integers) {
7         num1 = integers.get(0);
8         num2 = integers.get(1);
9     }
10
11    public double calculate(final char operator) {
12        switch (operator) {
13            case '+' : return num1 + num2;
14            case '-' : return num1 - num2;
15            case ':' : return Double.valueOf(num1) / Double.valueOf(num2);
16            case '*' : return num1 * num2;
17            case '%' : return num1 % num2;
18            default: return 0d;
19        }
20    }
21 }
```

Abbildung 4.23: Die Klasse CrazyCalculation

Je nach verwendetem Operator wird eine Aktion ausgeführt. Im gezeigten Beispiel wurde die Funktionalität minimal gehalten. Wenn neue Operationen hinzukommen (die bestehende Implementierung wird erweitert), so muss das Switch-Statement angepasst werden.

Im gezeigten Beispiel modifiziert man die Klasse CrazyCalculation wenn, neue Funktionalität hinzugefügt werden soll. So könnte man bei einer Division den Nenner auf 0 prüfen. Es wird an der entsprechenden Stelle Quellcode eingefügt. Entsprechend wird man für jede Case-Anweisung, Operationen in Methoden auslagern. Eventuell folgen Überlegungen, diese Funktionalität in eine Klasse zu kapseln. So wird über 6 Branches (Abbildung 4.24) eine entsprechende Klasse aufgerufen. Implementiert man weitere Funktionalitäten (radizieren, quadrieren, ...) so erhöhen sich die Branches, die getestet werden müssen. Dementsprechend nimmt die Komplexität der Tests zu.

Eine andere Lösung ist mit Polymorphismus zu erreichen. Anstelle eines Switch-Statement, wird für jeden Operator eine neue Klasse implementiert. So ist es möglich, ohne Änderungen bestehender Klassen, die Funktionalität zu erweitern.

In Abbildung 4.25 ist dies demonstriert. Alle zusätzlichen Erweiterungen implementieren das Interface *Operator*. Soll ein neuer Operator implementiert werden, so muss lediglich

```
22                  public double calculate(final char operator) {
23      1   6/6         switch (operator) {
24      1                   case '+' : return num1 + num2;
25      1                   case '-' : return num1 - num2;
26      1                   case ':' : return Double.valueOf(num1) / Double.valueOf(num2);
27      1                   case '*' : return num1 * num2;
28      1                   case '%' : return num1 % num2;
29      1                   default: return 0d;
30                      }
31                  }
32              }
```

Abbildung 4.24: Branches des Switch-Statements

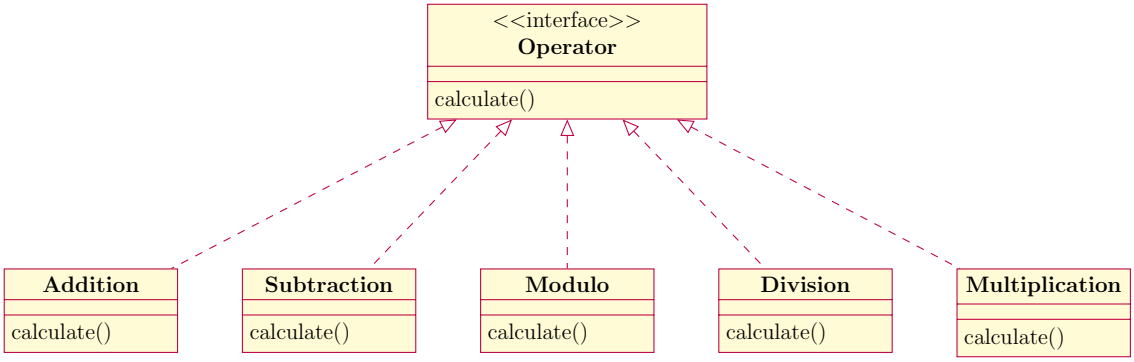

Abbildung 4.25: Taschenrechner mit Polymorphismus

eine neue Klasse erstellt werden, die das Interface *Operator* implementiert. Zwar werden nun mehr Klassen benötigt, jedoch lässt sich das Projekt einfacher erweitern.

Auch ist eine Verringerung der zu testenden Branches zu erwarten. Abbildung 4.26 verdeutlicht dies.

```
1  public class Calculators {
2
3      public Operator getCalculator(final char operator) {
4          Map<Character, Operator> operators = new HashMap<Character, Operator
               >();
5          operators.put('+', new Addition());
6          operators.put('-', new Subtraction());
7          operators.put(':', new Division());
8          operators.put('*', new Multiplication());
9          operators.put('%', new Modulo());
10         if (!operators.containsKey(operator)) {
11             return new UnknownOperator();
12         }
13         return operators.get(operator);
14     }
15 }
```

Abbildung 4.26: Die Klasse Calculators

Die zu testenden Branches wurden auf zwei Branches verringert. Es wurde ebenso dargestellt, dass der Einsatz eines Switch-Statements nicht zwingend erforderlich ist.

Zusammengefasst lässt sich sagen, dass eine Umsetzung mit Polymorphismus anstelle eines Switch-Statements mit Mehraufwand verbunden ist. Jedoch werden dadurch die Testklassen vereinfacht, sowie die Testbarkeit und die Erweiterbarkeit einer Klasse gesteigert.

Die Metrik Switch Density

Diese Metrik bewertet ein Switch-Statement, wenn dieses einen zu großen Umfang besitzt. So wird für jede Anweisung innerhalb eines Switch-Statements der Wert für Switch-Density inkrementiert.

Verfügbarkeit in Sonar

Die Metrik Switch-Density wird in Sonar zur Verfügung gestellt. Diese besitzt einen Standardwert von 10. Wenn also mehr als 10 Anweisungen innerhalb des Switch-Statements gefunden werden, so wird eine Violation ausgelöst. Dieser Wert kann als Basis für einen Verstoß gegen OCP verwendet werden.

Ergebnis

Es konnten bisweilen nur Switch-Statements als mögliche Verstöße gegen OCP gefunden werden. Durch ein Re-Design können diese durch den Einsatz von Polymorphismus ersetzt werden.

4.4.8 Testabdeckung Unittests

Die Testabdeckung von Unittests wurde bereits bei der Festlegung der Hauptziele des Qualitätsmodelles festgelegt.

Verfügbarkeit in Sonar

Für die Testabdeckung von Unittests steht in Sonar eine Übersicht zur Verfügung. Hierbei wird unterschieden zwischen Line-Coverage und Branch-Coverage. Es kann jede Klasse daraufhin untersucht werden, wofür eine separate Ansicht zur Verfügung steht. Sind in einem Projekt Unittests vorhanden, so werden diese automatisch erkannt und während der Sonar-Analyse ausgeführt. Es sind keine zusätzlichen Konfigurationen notwendig.

4.4.9 Testabdeckung Integrationstests

Die Testabdeckung von Integrationstests wurde bereits bei der Festlegung der Hauptziele des Qualitätsmodelles festgelegt.

Verfügbarkeit in Sonar

Für die Testabdeckung von Integrationstests steht ebenfalls eine Ansicht in Sonar zur Verfügung. Hierbei wird, wie bei den Unittests, in Line-Coverage und Branch-Coverage unterschieden. Es fehlt jedoch der Zusammenhang zwischen Erfüllungsgrad der Anforderungen und der Testabdeckung der einzelnen Klassen. So kann keine Schlussfolgerung daraus gezogen werden, wenn Integrationstests 50% Programmquelltext abdecken, dass dann auch nur die Hälfte der Anforderungen erfüllt sind.

4.4.10 Die Güte von Unittests

Im letzten Abschnitt der Definition von Metriken, soll die Güte von Unittests betrachtet werden. Das Qualitätsmodell wurde auf die Testbarkeit von Quellcode ausgelegt.

Auch wenn die Testabdeckung von Unittests 100% beträgt, so bedeutet dies nicht, dass Unittests automatisch qualitativ hochwertig sind.

Testqualitätsmetriken

Harry Sneed sagt in [Sne10, Seite 224] folgendes: „Eine echte Testqualität ist das Verhältnis gefundener Fehler zur Summe aller Fehler im System.“ Sneed erwähnt nicht, dass er sich damit auf System- oder Integrationstests bezieht. Eine Betrachtung von Unittests erfolgt nicht. Er erwähnt außerdem, dass gerade an Forschungsprojekten mit Techniken gearbeitet wird, wo künstliche Fehler eingebaut werden um die Testmethoden einer Bewertung zu unterziehen. Er sagt außerdem, dass in der Praxis diese Vorgehensweise seltenst vorzutreffen ist, aus Angst, eingebaute Fehler nicht wiederzufinden.

Folgende Testqualitätsmetriken werden von Sneed betrachtet:

- Testeffektivität
- Testvertrauen
- Testeffizienz
- Restfehlerwahrscheinlichkeit

Die genannten Metriken sollen jedoch nicht weiter untersucht werden, da alle Metriken von einer bestimmten Anzahl von gefundenen Fehlern ausgehen. Sind Fehler in Unittests, so lässt sich ein Projekt nicht kompilieren. Es ist eine Grundvoraussetzung, dass alle Unittests erfolgreich durchlaufen.

Mutation des Produktionscodes

Mit der Mutation[9] des Produktionscodes kann die Qualität von Unittests ermittelt werden. Dabei wird bestehender Quellcode minimal geändert und anschließend bestehende Unittests ausgeführt.

Wird ein Mutation in den Quellcode eingebaut und der oder die Unittests, die den betreffenden Quellcode testen, ausgeführt, so muss mindestens einer dieser Tests fehlschlagen. Ist dies nicht der Fall, gilt die Mutation als „erfolgreich“ injiziert und deutet auf eine Testschwäche hin.

Folgendes Beispiel soll dies verdeutlichen:

```
1 if (a <= b) {
2   // do something
3 }
4
5 if (a < b) {
6   // do something
7 }
```

Abbildung 4.27: Beispiel einer Mutation von Quellcode

Zeile 1 bis 3 stellen den originalen Quellcode dar. Ein Beispiel einer Mutation des Quellcodes ist von Zeile 5 bis 7 dargestellt. Hier wird eine Bedingung von <= auf < verändert.

Wenn für dieses Beispiel die Grenzen nicht ausreichend getestet werden, so wird eine Mutation erfolgreich injiziert werden können.

Ein Test gilt dann als bestanden, wenn unerwartete Exceptions auftreten oder Erwartungswerte nicht diesen entsprechen, wie es der Test fordert. Mit Hilfe dieser Technik ist es also möglich, Tests zu finden, die lediglich den Quellcode durchlaufen, dabei jedoch auf Prüfen von Rückgabewerten oder Zuständen von Objekten verzichten. Auch lassen sich ungetestete Schlüsselstellen mit Mutationen aufdecken.

[9] Mutation: Veränderung

Verfügbarkeit in Sonar

Sonar stellt keine Möglichkeiten für Mutationen bereit. Es befindet sich jedoch ein Plugin in der Entwicklung. Diese Plugin nennt sich *Sonar-Pitest-Plugin*[10] und liegt in der Version 0.1-SNAPSHOT vor. Da es noch kein Release dazu gibt, muss dies selbst kompiliert werden. Der Quellcode wird über ein Subversion-Repository[11] bereit gestellt. Diese Plugin nutzt die Messergebnisse des Tools PIT[12] und stellt diese in Sonar über eine Violation *Survived Mutant* bereit.

Ergebnis

Das Mutation-Testing soll in das Qualitätsmodell aufgenommen werden. Dabei soll dies kein Merkmal für die Testbarkeit einer Klasse sein, sondern soll als Merkmal der Erweiterbarkeit einer Klasse festgehalten werden. Sind keine Unittests einer Klasse vorhanden, so können auch keine Mutationen erzeugt werden. Daher ist die Einordnung des Mutation-Testings unter der Erweiterbarkeit einer Klasse nur sinnvoll.

4.5 Einordnung der Metriken

Eine Befragung und anschließende Auswertung ergab folgende Metriken, die in das Qualitätsmodell aufgenommen werden:

Metrik	Verfügbarkeit in Sonar	Ebene
CDh	nicht verfübar	Klasse
ACDh	nicht verfübar	Klasse
rACDh	nicht verfügbar	System
Coverage Unittests	verfügbar	Klasse
Coverage Integrationstest	verfügbar	Klasse
Duplicated Blocks	verfügbar	Klasse
VOD	nicht verfügbar	Klasse
Switch-Density	verfügbar	Klasse
LCOM4	verfügbar	Klasse
CCN2	nicht verfügbar	Klasse

Tabelle 4.6: Metriken, die für das Qualitätsmodell ausgewählt wurden

[10] siehe http://docs.codehaus.org/display/SONAR/Pitest
[11] Subversion-Repository: http://svn.codehaus.org/sonar-plugins/trunk/pitest
[12] PIT: http://pitest.org/

Metriken, die nicht in Sonar zur Verfügung stehen, müssen implementiert werden. Die Implementierung der einzelnen Metriken wird in Kapitel 5 diskutiert.

4.5.1 Verantwortlichkeiten

Für das Open-Closed-Principle, das Single-Responsibility-Principle sowie für
Law of Demeter wurden Metriken ausgewählt, die die genannten Kriterien am besten erfüllen. Dabei wurde gezeigt, dass solche Verstöße ebenfalls einen Einfluss auf die Testbarkeit einer Klasse haben können. Die Einstufung erfolgt jedoch auf der Basis der Verantwortlichkeiten.

4.5.2 Abhängigkeiten

Die betrachteten Metriken von Stefan Jungmayr beschreiben die Isolierbarkeit einer Klasse, wenn diese getestet werden sollen. Dabei werden Abhängigkeiten temporär aus dem System entfernt und anschließend eine Neuberechnung durchgeführt. In diesem Zusammenhang kann diese Art der Metriken als das Kriterium Abhängigkeit festgelegt werden. So können beispielsweise neu eingeführte Abhängigkeiten als testkritisch dem Entwickler bekannt gemacht werden, wenn diese einen bestimmten Wert überschreiten.

4.5.3 Testaufwand

Der Testaufwand soll mit Hilfe der Metriken Klassenkomplexität und Duplicated Blocks beschrieben werden. Dies wurde im vorangegangenen Abschnitt diskutiert. Dabei wurde festgestellt, dass die Höhe der Komplexität, sowie das Vorhandensein von Duplicated Blocks (bzw. die Komplexität von Duplicated Blocks), einen Mehraufwand in der Implementierung von Tests erfordert. Hierfür wurde das Kriterium Testaufwand bestimmt, dass beide Metriken vereinen soll.

4.5.4 Testbarkeit

Die Testbarkeit soll mittels Testaufwand, Abhängigkeiten und Verantwortlichkeiten zusammengefasst werden. Alle festgelegten untergeordneten Kriterien wurden bei der Ausarbeitung so eingestuft, dass diese die Testbarkeit einer Klasse beeinträchtigen.

4.5.5 Erweiterbarkeit

Die Erweiterbarkeit soll mit der Testbarkeit, mit der Testabdeckung von Unittests sowie der Testabdeckung von Integrationstests beschrieben werden. Zusätzlich wurde die Mutation von Quellcode als Merkmal zur Erweiterbarkeit hinzugefügt, da hinsichtlich der Testbarkeit, die Mutationen weniger aussagekräftig sind. Gibt es keine oder nur wenige Unittests, so gibt es auch keine oder nur wenige Mutationen.

Ist ein hohes Coverage von Unittests vorhanden, bedeutet dies nicht, dass alle Unittests qualitativ hochwertig sind. Es kann also bei einer hohen Anzahl von erfolgreich injizierten Mutationen die Erweiterbarkeit eines Projektes beeinträchtigt werden. So können neu implementierte Fehler unentdeckt bleiben.

4.5.6 Ergebnis

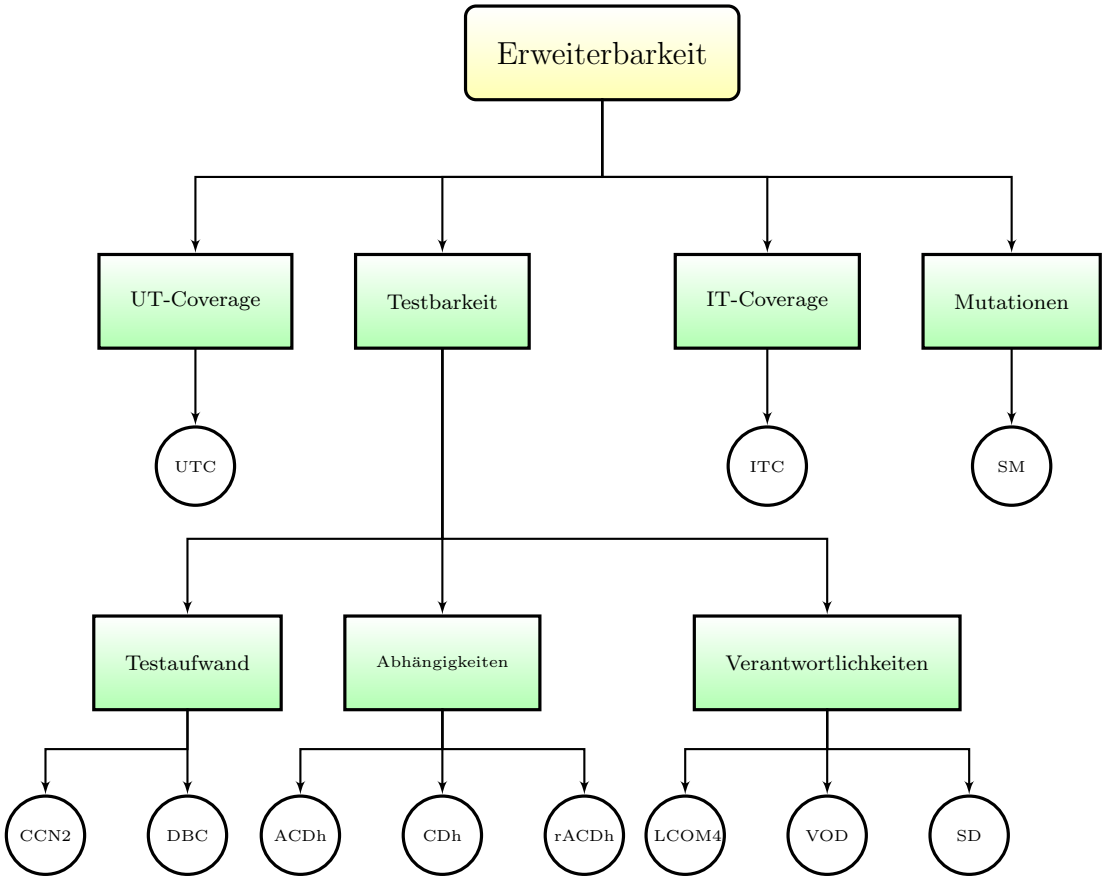

Abbildung 4.28: Qualitätsmodell Erweiterbarkeit - Metrikdefinitionen

Es wurden elf Metriken für das Qualitätsmodell ausgesucht, die die festgelegten Ziele und

Kriterien am besten beschreiben können. Dafür wurden Entwickler befragt und ebenfalls aus literarischen Quellen neue Ansätze in das Modell integriert.

Kapitel 5

Prozessintegration

5.1 Sonar als statische Quellcodeanalyse

Es soll folgend Sonar kurz vorgestellt werden. Hierbei sollen die wichtigsten Konzepte diskutiert werden, die auch in der Berechnung bzw. Darstellung des Qualitätsmodelles eine Rolle spielen. Alle weiteren Konzepte und Vorzüge von Sonar können auf der Homepage[1] von Sonar nachgelesen werden.

5.1.1 Das Violationkonzept von Sonar

Sonar stellt mit Hilfe der Tools PMD, Findbugs und Checkstyle eine Ansicht zur Verfügung, die alle gefundenen Fehler dieser Tools zusammenfasst. Dabei beziehen sich diese Fehler auf ineffizienten Code, toten Code, komplizierte Ausdrücke oder beispielsweise auf Formatierung des Quellcodes. Dabei gibt es fünf Einstufungen der Violations:

- Info
- Minor
- Major
- Critical
- Blocker

[1] http://www.sonarsource.org

Diese Einstufungen sind individuell einstellbar und können bei Bedarf abgeändert werden. So lässt sich jede Violation für jedes Projekt individuell konfigurieren. Das bringt den Vorteil, dass mit unterschiedlichen Projekten auch unterschiedliche Ziele verfolgt werden können. Hierzu müssen jedoch eigene Qualitätsprofile erstellt werden. Diese können jedoch voneinander abhängig sein.

Da Sonar schon länger im Einsatz ist und mit dieser Master-Thesis Sonar in den Entwicklungsprozess integriert werden soll, ist auch eine genauere Betrachtung der einzelnen verfügbaren Violations in Sonar von Nöten. Da die bereits voreingestellten Violations nicht ins Qualitätsmodell aufgenommen wurden, ist hierfür eine separate Betrachtung erforderlich. Dazu wurde ein Forum eröffnet, wo alle Violations und deren Einstufungen diskutiert werden können.

Jede Violation kann ebenfalls als False-Positiv[2] markiert werden. Ebenso können neue Violations generiert werden, die dann einem entsprechenden Kontext bzw. einer Metrik zugeordnet werden können. Dies entspricht einer Positiv-False[3] Festlegung.

5.1.2 Reviewplanung in Sonar

Mit Hilfe von Violations lassen sich verschiedene Reviews mit Sonar planen. Hierzu können verschiedene Action-Plans aufgestellt werden, die verschiedene Reviews beinhalten können. Wenn also eine *Qualitätsverbesserung* eines Projektes durchgeführt werden soll, dann kann jede Violation einem Action-Plan zugeteilt werden. Zur Reviewplanung kann also ein Team zusammengeführt werden und entsprechende Violations hierfür zusammenfassen, die dann entfernt werden sollen. Jede Violation lässt sich hierbei einem einzelnen Entwickler zuteilen. Dabei kann die Kommunikation des Teams gesteigert werden und ebenso werden sich einzelne Entwickler verstärkt mit Sonar auseinandersetzen. Auch lässt sich gemeinsam besser diskutieren, ob entsprechende Violations Falschaussagen sind oder bestimmte Codezeilen als beispielsweise *testkritisch* eingestuft werden können, wenn Metriken an den entsprechenden Stellen versagt haben.

5.1.3 Notification-Channels von Sonar

Sonar stellt standardmäßig einen Email-Channel zur Verfügung. Hier kann festgelegt werden, über welche Events man benachrichtigt werden möchte. Dabei fehlt bisweilen die

[2] die entsprechende Violation wird als eine Falschaussage festgelegt
[3] dies entspricht einer Violation, die an entsprechender Stelle nicht gefunden wurde

Benachrichtigung über neu erstellte Violations innerhalb eines einzelnen Commits. Hierfür wurde das Plugin erweitert, das diese entsprechende Funktion zur Verfügung stellt. Dabei wird jedoch nicht der Verursacher in dieser Email aufgeführt, sondern lediglich die Anzahl der neu generierten Violations numerisch dargestellt. Ein zusätzlicher Link verweist auf das Sonarprojekt.

5.2 Bewertung der Qualitätsmerkmale

Die Bewertung der Qualitätsmerkmale erfolgt unter der Betrachtung der Implementierung des Qualitätsmodelles. Dabei soll ein eigenständiges Plugin für Sonar entwickelt werden, das alle Kriterien des Modells berücksichtigt. Auf Implementierungsdetails soll nicht eingegangen werden, sondern nur die wesentlichen Punkte diskutiert werden.

5.2.1 Testaufwand

Der Testaufwand soll mit Hilfe der McCabe-Metrik ausgedrückt werden. Dabei wurde festgehalten, dass private Methoden keinen Grundwert erhalten und Duplicated Blocks in ihrer Komplexität bewertet werden sollen.

Klassenkomplexität

Die Berechnung der Komplexität in Sonar erfolgt auf der Basis folgender Elemente und wird für diese Elemente inkrementiert:

- if
- for
- while
- case
- catch
- throw
- return (vorzeitiger Methodenrücksprung)

Jede Methode (außer Getter- und Settermethoden) wird zusätzlich mit einem Grundwert von Eins versehen. Die Klassenkomplexität wird aus der Summe der Komplexitäten einer jeden Methoden errechnet. Mit der zusätzlichen Forderung, private Methoden ohne Grundwert zu versehen, wird auf Basis der Klassenkomplexität eine angepasste Klassenkomplexitätszahl errechnet. Hierzu wird die Metrik *Public API* verwendet, die die Anzahl der öffentlichen Methoden einer Klasse zählt. Ebenso wird die Metrik *Functions* verwendet, die die gesamte Anzahl der Methoden zur Verfügung stellt, mit Ausnahme von Getter- und Settermethoden.

Zur Berechnung der angepassten Klassenkomplexität ergibt sich folgende Formel:

$$CCN2 = CCN - (Functions - PublicAPI)$$

$$\text{für } Functions - PublicAPI >= 0$$

Komplexität von Duplicated Blocks

Das Aufdecken von doppeltem Quellcode kann von einer proprietären CPD-Engine von Sonar durchgeführt werden oder über die CPD-Engine von PMD. Da die Entwickler von Sonar relativ kurze Releasezyklen bereitstellen, ist die Verwendung der proprietären CPD-Engine vorzuziehen.

Der Zugriff auf die Metrik *Duplicated Blocks* in Sonar erfolgt über eine XML-Struktur, die in der Datenbank von Sonar abgespeichert ist. In dieser XML-Struktur wird für jede Ressource[4] eine solche XML-Datei zur Verfügung gestellt. In ihr sind alle *Duplicated Blocks* enthalten, die ressourcenübergreifend gefunden wurden. Dabei dient diese XML-Struktur lediglich zur Darstellung des Codeabschnittes im Sonar-Webinterface. Es werden jedoch Informationen zu Klassenname, sowie Anfangs- und Endzeile zur Verfügung gestellt. So lassen sich diese Codezeilen auf Komplexität untersuchen.

Wird ein *Duplicated Block* gefunden, so wird aus den entsprechenden Ressourcen mittels eines JavaParsers[5] ein Abstract Syntax Tree[6] (AST) generiert. Mit Hilfe des AST ist es nun möglich, die Komplexität dieses Codeabschnittes zu berechnen. Dabei wurden gleiche Berechnungsvorschriften, die auch für die Berechnung der Methodenkomplexität von Sonar durchgeführt werden, berücksichtigt. Ebenso werden private Methoden nicht mit einem Grundwert von Eins versehen.

[4] oder auch Java-Datei
[5] siehe dazu http://code.google.com/p/javaparser/
[6] siehe dazu http://en.wikipedia.org/wiki/Abstract_syntax_tree

Die Komplexität von *Duplicated Blocks* wird mittels der Abkürzung DBC festgelegt. Der Testaufwand errechnet sich dann aus der Formel:

$$Testaufwand = CCN - (Functions - PublicAPI) + DBC$$

Darstellung des Testaufwandes

Für die Darstellung des Testaufwandes wurde das Konzept der Violations von Sonar übernommen.

Dabei kann für das Teilmerkmal Testaufwand eine Grenze festgelegt werden, ab wann eine neue Violation generiert werden soll. Zusätzlich zur Festlegung der Grenze kann der Aufwand zur Beseitigung dieser Violation festgelegt werden. Dies kann individuell für jedes Projekt über das Webinterface von Sonar konfiguriert werden. Für eine Reviewplanung fehlt bisweilen die Festlegung des Aufwandes individuell für jede einzelne Violation. So stellt der Standardwert nur eine grobe Näherung dar. Es kann jedoch damit gerechnet werden, dass in Zukunft auch dieser Wert für jede Violation individuell festgelegt werden kann.

5.2.2 Abhängigkeiten

Die Metrik CDh

Wie bereits angesprochen, werden Metriken, die für die Berechnung der Reduktionsmetrik *rACDh* notwendig sind, in Sonar nicht zur Verfügung gestellt. Hierfür ist es also notwendig, mittels des Plugins eine Implementierung der entsprechenden Metriken vorzunehmen.

Für die Metrik CDh wurde dabei der JavaParser[7] von Google eingesetzt, indem auch hier ein AST generiert wurde. Aus diesem generierten AST wurde anschließend ein vereinfachter Abhängigkeitsgraph in einer separaten Datenstruktur festgehalten. Hierbei wurden jedoch nur Klassen aus dem entsprechenden Projekt berücksichtigt. Klassen, die nicht Teil des Projektes sind, haben demnach keinen Einfluss auf die CDh-Metrik und sollen unberücksichtigt bleiben.

Mit Hilfe des vereinfachten Abhängigkeitsgraphen lässt sich die CDh-Metrik einer jeden Klasse ermitteln und festhalten.

[7] siehe dazu http://code.google.com/p/javaparser/

Die Metrik ACDh

Die Metrik ACDh wird aus dem vereinfachten Abhängigkeitsgraphen erfasst. Hierfür wird der komplette Abhängigkeitsgraph durchlaufen und die bereits erfassten CDh-Metriken aufsummiert.

Die Reduktionsmetrik rACDh

Die Reduktionsmetrik wird ebenfalls aus dem vereinfachten Abhängigkeitsgraphen erfasst. Hierzu werden für jede Klasse einzeln die Abhängigkeiten entfernt, die mittels Konstruktoraufrufe erzeugt werden. Anschließend wird der neu erzeugte Abhängigkeitsgraph mit der ACDh-Metrik bewertet. Daraufhin wird die Differenz aus dem tatsächlichen ACDh-Wert mit dem temporären ACDh-Wert ermittelt und überprüft. Übersteigt die Differenz die konfigurierbare Größe, so wird eine neue Violation generiert, die ebenfalls mit einem konfigurierbaren Aufwand gespeichert wird.

Die Darstellung dieser textkritischen Abhängigkeit erfolgt über eine generierte Violation. Es wird zusätzlich die Differenz und der CDh Wert dieser Abhängigkeit mit der generierten Violation abgespeichert und dargestellt.

5.2.3 Verantwortlichkeiten

Law of Demeter

Für Law of Demeter wurde bisweilen in Sonar keine Metrik zur Verfügung gestellt. PMD bietet in der Version 5.0-ALPHA jedoch eine Messung zu VOD an. Da jedoch Sonar aktuell mit der PMD-Version 4.3 arbeitet, ist auch über PMD keine Metrik zu VOD verfügbar.

Aus diesem Grund wurde untersucht, ob der Einsatz von PMD-5.0 in Sonar möglich ist. Hierzu wurde der aktuelle Quellcode von PMD kompiliert und in einem lokalen Maven-Repository zur Verfügung gestellt.

Für ein erfolgreiches Kompilieren des Sonar-Plugins ist die aktuelle Version von PMD notwendig und wird momentan noch nicht über ein öffentliches Maven-Repository bereitgestellt.

Die Ausführung von PMD in der Version 5.0-SNAPSHOT mit nur einer zu berücksichtigenden Regel[8] erwies sich als unbenutzbar. Die Implementierung der Metrik Law

[8] siehe http://pmd.sourceforge.net/snapshot/xref/net/sourceforge/pmd/lang/java/rule/coupling/LawOfDemeterRule.html

of Demeter in PMD zeigt zwar eine Teilmenge zur Metrik VOD, jedoch wird eine größere Menge an Falschaussagen generiert.

Die Betrachtung der Implementierung ergab, dass die Einhaltung der 4 Gesetze von Demeter nicht berücksichtigt wurden. Aus diesem Grund wurde eine Neuimplementierung durchgeführt, die jedoch auf der Implementierung von PMD aufbaut.

Das Open-Closed-Principle

Für OCP wurde die Metrik Switch-Density ausgewählt. Diese Metrik beschreibt den Umfang eines Switch-Statements, der maximal erlaubt ist. Die zu berücksichtigende Größe eines Switch-Statements kann über das Webinterface von Sonar konfiguriert werden. Ebenso kann ein Aufwand festgelegt werden, diesen Verstoß zu beseitigen.

Das Single-Responsibility-Principle

SRP soll mit der Metrik *LCOM4* ausgedrückt werden. Sonar bietet diese Metrik bereits an und stellt zusätzlich eine Ansicht für diese Metrik zur Verfügung.

Die Implementierung erfolgt durch die Überprüfung jeder Klasse auf den LCOM4-Wert. Wird eine Klasse gefunden mit einem Wert für LCOM4 > 1, dann wird eine Violation für diese Klasse generiert. Zusätzlich zu dieser Violation wird ein konfigurierbarer Aufwand gespeichert, der die Beseitigung dieses Verstoßes ausdrücken soll.

5.2.4 Testbarkeit

Die Testbarkeit wird aus den Teilmerkmalen

- Testaufwand
- Abhängigkeiten
- und Verantwortlichkeiten

ermittelt. Hierzu werden einerseits die Anzahl der Verstöße ermittelt, andererseits die anfallenden Kosten, jene Verstöße zu beseitigen. Zusätzlich wird auf Systemebene die Testbarkeit ermittelt, die sich wiederum aus der Summe der erfassten Kosten errechnet, dividiert durch die Anzahl der Klassen. Dies soll einen Vergleich von Projekten ermöglichen.

Auf Systemebene wird ein Drilldown der Violations zur Verfügung gestellt, der eine genaue Betrachtung der Verstöße auf Klassenebene berücksichtigt. Ebenso wird die Zusammensetzung der Testbarkeit aus den einzelnen Teilmerkmalen zur Verfügung gestellt. Hierzu werden ebenfalls die Kosten auf Systemebene ermittelt. Auch hier wird ein Drilldown der Verstöße zur Verfügung gestellt.

5.2.5 Mutationen des Produktionscodes

Die Mutationen des Produktionscodes werden unabhängig der Testbarkeit einer Klasse ermittelt. Mit Hilfe der Qualität von Unittests soll ausschließlich die Erweiterbarkeit einer Klasse bzw. eines Projektes ausgedrückt werden. Hierfür soll das Sonar-Pitest-Plugin Verwendung finden. Für dieses Plugin steht momentan noch kein Release zur Verfügung. Ein Zugriff auf ein Versionsverwaltungssystem wird jedoch zur Verfügung gestellt.

Dieses Plugin stellt die Ergebnisse des Frameworks PIT bereit und generiert für jede erfolgreich injizierte Mutation eine neue Violation.

5.2.6 Erweiterbarkeit

Die Erweiterbarkeit errechnet sich aus der Testbarkeit, sowie den Mutationen des Produktionscodes. Das Coverage von Unit-Tests sowie das Coverage von Integrationstests wird nicht in die Bewertung der Erweiterbarkeit aufgenommen. Diese zwei Merkmale dienen ausschließlich der Betrachtung auf Klassenebene und dabei soll entschieden werden, ob Unittests nachgepflegt werden können oder der Einsatz von Integrationstests sinnvoller erscheint. Diese Betrachtung ist demnach nur für bestehende Projekte (oder auch Legacy-Projekte) sinnvoll, wenn eine Qualitätsverbesserung des Projektes durchgeführt werden soll.

Bei einer Neuentwicklung eines Projektes spielen diese Betrachtungen eher eine untergeordnete Rolle. Wird das Projekt mittels TDD entwickelt, so wird automatisch ein höheres Coverage der Unittests erreicht. Dementsprechend muss nicht entschieden werden, ob Unittests oder Integrationstests implementiert werden müssen.

5.2.7 Darstellung der Messergebnisse

Die Darstellung der Messergebnisse soll über das Plugin separat bereitgestellt werden. Dazu lässt sich das Dashboard von Sonar erweitern, indem die dafür entsprechende API

verwendet werden kann.

Darstellung der Violations

Die Darstellung der Violations erfolgt über ein bereits vorhandenes Widget von Sonar. Alle zusätzlich eingeführten Violations werden in dieser Ansicht ebenfalls mit aufgenommen.

Darstellung der Qualitätsmerkmale

Die Darstellung der Qualitätsmerkmale soll über ein separates Widget erfolgen. Dazu wird eine Übersicht auf Systemebene und ebenso ein Drilldown bis auf Klassenebene bereitgestellt.

Auf Systemebene wird

- Erweiterbarkeit
- Testbarkeit
- Testaufwand
- Abhängigkeiten
- Verantwortlichkeiten

mit der Summe aus den anfallenden Kosten durch die Anzahl der Klassen zur Anzeige gebracht. Ein Drilldown bis zu jeder einzelnen Violation und den anfallenden Kosten wird zur Verfügung gestellt.

Zusätzlich soll ein Diagramm einen groben Überblick des Zustandes des Projektes vermitteln. Dafür soll die bereits zur Verfügung stehende API von Sonar verwendet werden. Hier bietet sich ein Netzdiagramm an, mit dem Ausprägungen einzelner Merkmale bzw. Metriken dargestellt werden können.

5.3 Ausführung einer Sonaranalyse

Die Ausführung einer Sonaranalyse soll kurz betrachtet und diskutiert werden. Es stellt sich die Frage, in welchen Abständen eine Ausführung einer Sonaranalyse stattfinden soll und wie diese durchgeführt werden kann.

5.3.1 Architektur von Sonar

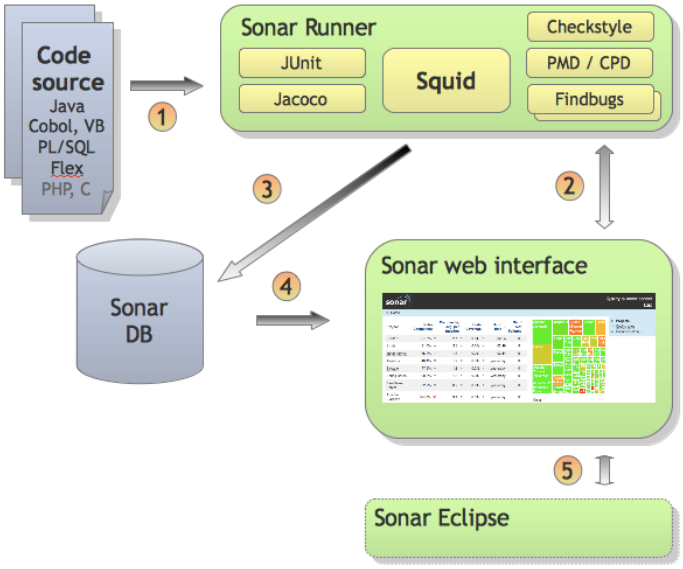

Abbildung 5.1: Architektur von Sonar (Quelle: http://docs.codehaus.org/display/SONAR/Install+Sonar)

Die Ausführung der Sonaranalyse kann über mehrere Wege erfolgen. Mit Apache Maven[9], Apache Ant[10] oder dem Standalone-Tool Sonar-Runner[11]

In dieser Abteilung wird mit Apache Maven gearbeitet, daher soll auch eine Analyse mit Apache Maven durchgeführt werden. Die Analyse muss in Punkt **1** ausgeführt werden. Dies kann lokal auf jeder Entwicklerstation oder über einen CI-Server erfolgen.

5.3.2 Lokale Ausführung auf Entwicklerstation

Eine Sonaranalyse kann von jeder Entwicklerstation ausgeführt werden. Dabei kann auf eine lokale oder zentrale Datenbank von Sonar zugegriffen werden. Hierbei muss für die Ausführung mit Apache Maven ein Profil angelegt werden, indem die URL von Sonar und ebenfalls der Zugriff auf eine Datenbank festgehalten wird. Mit

```
mvn sonar:sonar -PmeinProfil
```

[9] siehe http://maven.apache.org/
[10] siehe http://ant.apache.org/
[11] siehe http://docs.codehaus.org/display/SONAR/Analyse+with+a+simple+Java+Runner

wird eine dementsprechende Analyse gestartet.

Es kann jedoch der Entwickler selbst entscheiden, wann er eine Analyse starten möchte.

5.3.3 Ausführung auf CI-Server

Die Ausführung einer Sonaranalyse auf einem CI-Server stellt eine bessere Variante dar, ein dauerhaftes Messverfahren einzuführen. In dieser Abteilung wird bereits mit dem CI-Server Jenkins gearbeitet. Es fehlte bisweilen jedoch eine Ausführung einer Messung von Sonar.

Die Jenkins-Pipeline

Für Jenkins kann über das Plugin Build-Pipeline eine Build-, Test-, und Deployment-Automation eingeführt werden. Innerhalb einer Pipeline können mehrere Jobs konfiguriert werden. So ist es möglich einen Buildprozess, einen Testprozess, einen Releaseprozess und auch einen Deploymentprozess in genannter Reihenfolge automatisch auszuführen. Ebenso können bei Fehlschlagen einzelner Prozesse keine weiteren nachfolgenden Prozesse ausgeführt werden. So lässt sich in dieser Pipeline ebenfalls eine Sonaranalyse integrieren, die ebenso fehlschlagen sollte, wenn eine entsprechende Verschlechterung des Quellcodes festgestellt wurde.

Ein Nachteil besteht bei dieser Variante der Messung. So wird für jeden Commit eine neue Analyse ausgeführt. Bei entsprechend umfangreichen Projekten, kann die Durchführung einer Sonaranalyse viel Zeit in Anspruch nehmen. So wird momentan für das Projekt Tristan, bestehend aus 5214 Klassen und 288158 LOC, 30 Minuten Zeit in Anspruch genommen. Wenn also der Build-Prozess unterbrochen wurde, so müssen zunächst die aufgedeckten Qualitätsmängel am Quellcode beseitigt werden. Ist ein Release geplant bzw. erforderlich, so kann hier ein Zeitverzug entstehen.

5.3.4 Verschlechterung des Quellcodes

Eine Verschlechterung des Quellcodes soll anhand neuer Violations festgestellt werden. Werden neue Violations hinzugefügt, so wird der Analyse-Prozess in der Jenkins-Pipeline als ein Fehlschlag markiert. Erst wenn die entsprechenden Violations entfernt wurden, können darauf folgende Prozesse wieder ausgeführt werden.

Dieses Konzept wurde auf Metriken ausgedehnt. So kann individuell festgelegt werden, welche Metriken unter Beobachtung stehen sollen. Auch können Violations individuell festgelegt werden.

Als Standard werden Violations die als

- Major
- und Blocker

eingestuft sind überprüft. Ebenso ist die Metrik

- Unittest-Coverage

als Standard deklariert wurden.

5.3.5 Benachrichtigung per Email

Wird eine Verschlechterung festgestellt, so kann individuell festgelegt werden, ob und welche Entwickler eine Email erhalten sollen. So kann auch nur der Erhalt einer Email erfolgen, sodass die Entwickler auch weitere Prozesse innerhalb der Pipeline ausführen dürfen.

5.3.6 Export aller Messungen von Sonar

Um die anfallenden Messungen aus der Sonar-Datenbank exportieren zu können, wurde ein CSV-Export implementiert. Dabei werden alle Metriken zu denen es Messungen gibt in eine CSV-Datei exportiert. Dabei steht jede Zeile für eine separate Messung.

Mit Hilfe des CSV-Exports können später statistische Untersuchungen durchgeführt werden, mit deren Hilfe beispielsweise eine externe Validierung des Qualitätsmodelles durchgeführt werden kann.

5.4 Einbeziehung der Entwickler

5.4.1 Sonar-Forum

Für die eingesetzten Metriken sollen die Entwickler ein Mitspracherecht haben. Hierfür wurde ein Forum eingerichtet, indem alle Metriken diskutiert werden können. Momentan

werden dabei jedoch ausschließlich die vorhandenen Violations von Sonar diskutiert. So können eventuell abgeschaltete Violations eingeführt oder vorhandene Violations entfernt werden. Eine Umfrage soll diese letzten Endes ermitteln, wenn keine eindeutige Lösung gefunden wurde. Während der Bearbeitungszeit dieser Master-Thesis, sind so einige interessante Diskussionen und Umfragen entstanden, mit dem Ergebnis, dass Violations hinzugefügt und auch entfernt wurden.

Kapitel 6

Fallstudie

Im folgendem Kapitel soll eine Fallstudie durchgeführt werden. Dabei wird das entwickelte Plugin zum Einsatz kommen und in den Entwicklungsprozess integriert.

Untersucht werden soll, ob eine Sonaranalyse nach jedem Commit durchgeführt werden muss. Damit verbunden, ob ein Abbruch eines Builds dem Entwickler Vorteile bringt.

Auch sollen die einzelnen Metriken betrachtet und Einschätzungen abgegeben werden. Die Betrachtung der einzelnen Metriken und einem damit verbundenen Refactoring, soll mittels Pair-Programming[1] durchgeführt werden.

6.1 Integration des Plugins in den Entwicklungsprozess

Die Integration des Plugins in den Entwicklungsprozess erfolgte bereits sechs Monate vor der Durchführung dieser Fallstudie. Darin enthalten war lediglich der Abbruch des Build-Prozesses innerhalb der Jenkins-Pipeline.

Die ausgearbeiteten Metriken sollten für diese Fallstudie nun ebenfalls integriert werden. Die Akzeptanz dessen war jedoch sehr gering, sodass die neuen Metriken zunächst nicht integriert wurden bzw. wieder aus den Messungen entnommen wurden. Dabei waren alle ausgearbeiteten Metriken auf INFO eingestellt, sodass zunächst der Build-Prozess nicht unterbrochen wurde. Lediglich die Analysezeiten sind gestiegen, was von den Entwicklern bemängelt wurde. Außerdem wurde der Wunsch mehrfach geäußert, selber entscheiden zu dürfen, ob und wann die neuen Metriken für ihre Projekte eingesetzt werden sollen.

[1] siehe http://www.extremeprogramming.org/rules/pair.html

Es konnte also nicht die komplette Abteilung dazu animiert werden, an dieser Fallstudie teilzunehmen. Lediglich vier Entwickler waren mit insgesamt fünf Projekten daran beteiligt.

6.2 Abbruch des Build-Prozesses

Bei der Erzeugung neuer Violations (Major, Critical, Blocker) wird der Build-Prozess abgebrochen (je nach Konfiguration).

Daraus entstanden zunächst Anfragen, eine lokale Sonar-Instanz auf den Entwicklerstationen bereitzustellen. Jeder Entwickler führte dann zunächst eine lokale Analyse vor einem Commit durch um ein Fehlschlagen des Builds auf dem CI-Server zu verhindern.

Daraus folgten mehrere Probleme:

- Synchronisation der Einstellungen von Sonar
- Fehlende Integration in eine Entwicklungsumgebung (Eclipse, IntelliJ)
- Verzögerungen von Commits

Durch die teilweise langen Analysezeiten von Sonar zeigte sich eine Ablehnung dessen. Auch wurde darauf hingewiesen, wenn Auslieferungen bevorstehen, dass zusätzlicher Aufwand besteht bzw. der Stressfaktor des Entwicklers erhöht wird.

Der Build-Prozess des Projektes Tristan benötigt 30 Minuten ohne Mutation-Testing. Mit aktiviertem Mutation-Testing bewegt man sich jenseits von 120 Minuten (Projekt Tristan besteht aus ca. 300.000 LOC). Die Analysezeit erhöht sich entsprechend mit zusätzlichen Unittests. So kann nicht innerhalb der Pipeline mit dem Mutation-Testing gearbeitet werden, sondern muss in einer separaten Analyse durchgeführt werden.

Es ist also notwendig, die Analysezeit von Sonar zu verkürzen. So könnte man beispielsweise nur die Klassen einer Analyse unterziehen, die neu hinzugefügt oder verändert wurden. Alle anderen Messergebnisse können aus einer vorherigen Messung übernommen werden. Dies würde die Anlysezeit von Sonar stark verkürzen, ist jedoch im Moment nicht möglich, da diese Funktionalität von Sonar nicht zur Verfügung gestellt wird.

Betrachtung der Metriken vor und während der Aktivierung des Plugins

Die Tabelle 6.2 zeigt die Messergebnisse des Projektes *Tristan* vor und während der Aktivierung des Plugins. Zwischen der Messung 2 und Messung 3 wurde das Plugin aktiviert und zwischen der Messung 3 und der Messung 4 wurde zusätzlich die Metrik Duplicated Blocks aufgenommen. Standardmäßig sind die Metriken *Blocker Violations, Critical Violations, Major Violations* und *Coverage* enthalten.

Metrik	Messung 1	Messung 2	Messung 3	Messung 4
LOC	268.179	277.283	296.472	285.968
Violations	17.554	16.615	11.838	8.478
Blocker Violations	0	0	0	0
Critical Violations	771	561	247	156
Major Violations	5.814	5.698	4.136	2.973
Coverage	26,2	27	31,3	35,9
Duplicated Blocks	2.072	2.112	2.220	1.494

Tabelle 6.1: Werte der Metriken vor und während der Aktivierung des Plugins

Hier zeigt sich nun, dass zuvor zwar keine zusätzlichen Violations hinzugekommen sind, jedoch seit der Aktivierung nun mehr Violations entfernt wurden als vorher. Ebenso zeigen die Messergebnisse der Metrik Duplicated Blocks, dass nun ebenfalls diese Metrik berücksichtigt wird. Auch werden nun vermehrt Unittests implementiert. Die Qualität dieser Unittests kann mit der Metrik Survived Mutant gemessen werden. Die Messungen zeigen, dass durchaus die Entwickler dazu animiert werden können, sich mit Sonar auseinander zu setzen, um mit Metriken zu arbeiten.

Fazit

Für kleine Projekte (< 30.000 LOC) ist die Analysezeit von <= 10 Minuten noch vertretbar [2]. Für größere Projekte sollte in Stresssituationen[3] auf die Sonaranalyse verzichtet werden. Dies gilt immer mit Absprache des Verantwortlichen des Projektes und des Abteilungsleiters. Die Ausführung einer Analyse nach jedem Commit und einem damit verbundenen Abbruch eines Build-Prozesses bringt zunächst die Vorteile, dass sich Entwickler mit Sonar auseinandersetzen. Auch zeigt der Einsatz dessen, dass entsprechend auf die Metriken reagiert wird. Ein großer Nachteil sind jedoch die Analysezeiten von Sonar.

[2] ohne Berücksichtigung eingesetzter Hardware
[3] Auslieferungen etc.

6.3 Betrachtung der einzelnen Metriken

Folgend werden einzelne Metriken näher betrachtet. Für die Betrachtung einzelner Metriken wurden zunächst Messungen mit dem Plugin durchgeführt. Nach erfolgreicher Messung wurden mittels Pair-Programming zunächst alle Violations eines Projektes betrachtet. Festgestellte Falschaussagen wurden als False-Positiv festgelegt. Aus den übrigen Ergebnissen wurden Stichproben entnommen und ein Refactoring durchgeführt.

Ebenfalls wurde die Testbarkeit einer Klasse untersucht. Eine Anforderung für Legacy-Projekte war es, Klassen zu finden, die einfach mit Unittests erweitert werden können.

6.3.1 Die Testbarkeit einer Klasse

Für die Testbarkeit einer Klasse werden der Testaufwand, die Reduktionsmetrik rACDh und die Verantwortlichkeiten zusammengefasst. Werden Violations für diese Kriterien generiert, so wird auch die Testbarkeit einer Klasse schlechter.

Folglich soll untersucht werden, ob Klassen die keine Violations enthalten, einfach zu testen sind und Klassen die Violations enthalten, schwerer zu testen sind.

Durch die begrenzte Zeit dieser Fallstudie wurde sich auf die Aussagen der Entwickler beschränkt. Eine Implementierung von Unittests erfolgte nicht.

Klassen ohne Violations

Es wurden keine Klassen in dieser Fallstudie gefunden, die sich auch ohne Violations schwer testen ließen. So kann gesagt werden, dass diese Anforderung durchaus als erfüllt betrachtet werden kann. Ausschlaggebend hierfür war die angepasste McCabe-Metrik. Je höher der Wert der McCabe-Metrik war, desto höher wurde ebenfalls der Testaufwand.

Es wurde bemängelt, dass eine Gegenüberstellung der Testbarkeit zur Testabdeckung von Unittests suboptimal ist. Die Testbarkeit und das Coverage von Unittests kann nicht direkt gegenübergestellt werden.

So muss jede Klasse, die keine Violations beinhaltet, betrachtet werden. Erst bei der Betrachtung der einzelnen Klasse kann das Coverage von Unittests abgelesen werden.

Klassen mit Violations

Klassen die Violations enthielten waren nicht immer schlecht testbar. Die Metriken VOD und rACDh zeigten hier eine Häufung von Falschaussagen.

6.3.2 Die Metrik VOD

Die Metrik VOD soll nun betrachtet werden. Dabei wurden mehrere Refactorings durchgeführt. Nach einem Refactoring wurde mit Sonar eine neue Messung ausgeführt und die Ergebnisse betrachtet. Dabei standen ebenfalls alle anderen Metriken des Modells unter Beobachtung.

Law of Demeter und fremde Bibliotheken

Werden bei der Benutzung von fremden Bibliotheken Violations zu Law of Demeter erzeugt, kann ein entsprechendes Refactoring *Extract Method*[4] Violations zu Law of Demeter beseitigen. In diesem Fall wird jedoch nicht immer die Testbarkeit einer Klasse verbessert.

Die folgende Methode aus einer Klasse zeigt dazu ein Beispiel, wodurch der Testaufwand dieser Klasse etwas verringert werden konnte.

```
1     public Id readId(final int i) throws SQLException {
2         String value = resultSet.getString(i);
3         if (null != value && 0 < value.length()) {
4             return new Id(value);
5         }
6         return null;
7     }
```

Abbildung 6.1: Methode readId

In Zeile 3 wird eine Violation zu Law of Demeter erzeugt. Führt man an dieser Stelle ein Refactoring *Extract Method* durch, so kann diese Funktionalität in eine neue Methode ausgelagert werden. In Abbildung 6.2 ist dies dargestellt.

[4]Funktionalität in eine Methode auslagern

```
1     public Id readId(final int i) throws SQLException {
2         String value = resultSet.getString(i);
3         if (isSet(value)) {
4             return new Id(value);
5         }
6         return null;
7     }
```

Abbildung 6.2: Methode readId refaktorisiert

Die Methode *isSet* wird nun von weiteren vier Methoden verwendet. So wurde die Komplexität der Klasse von 22 auf 18 verringert und es konnten dementsprechend Tests gelöscht werden. Damit konnte der Testaufwand etwas verringert werden.

```
1     private boolean isSet(final String value) {
2         return null != value && isSet(value.length());
3     }
4
5     private boolean isSet(final int value) {
6         return 0 < value;
7     }
```

Abbildung 6.3: Extrahierte Methode isSet

Law of Demeter und Methodenparameter

Folgend soll ein Refactoring vorgestellt werden, indem Parameter einer Methode verändert wurden, um eine Violation von LOD zu entfernen. Gleichzeitig wurde die dazugehörige Testklasse betrachtet.

```
1         passwordAspects.clear();
2         if (engine != null) {
3             PasswordAspectConfig config = engine.getMandantConfiguration().
                  getPasswordAspectConfig();
4             if (config != null) {
5                 passwordAspects.addAll(config.getPasswordAspects());
6             }
7         }
8
9         if (passwordAspects.isEmpty()) {
10            setupDefaultPasswordAspects();
11        }
```

Abbildung 6.4: Violations zu LOD

Die Abbildung 6.4 zeigt in Zeile 3 zwei Violations und in Zeile 5 eine Violation zu LOD. In Anhang B.1.1 ist die vollständige Klasse dargestellt. Ziel war es, diese Violations zu entfernen und die Auswirkungen zu betrachten.

Die Klassenvariable *engine* wurde in dieser Klasse nur an der Stelle, wie in Abbildung 6.4 zu sehen ist, verwendet. So kann die Klasse PasswordAspectConfig per Konstruktorinjection als Klassenvariable initialisiert werden. Eine vorherige Testanpassung konnte nicht durchgeführt werden, da keine Testklasse zur Verfügung stand. Daher wurde zunächst eine Testklasse erstellt. Da die Klasse ClientEngine ein aufwendigeres Setup benötigte, wurde auf eine Testimplementierung vor dem Refactoring verzichtet. Daher wird bereits ersichtlich, dass die Metrik VOD hier richtig lag. Auch muss im Testsetup MandantConfiguration bereitgestellt werden.

In der Testklasse ist nun lediglich das Bereitstellen einer Instanz zur Klasse PasswordAspectConfig notwendig.

```
1
2     private void setupPasswordAspects() {
3         passwordAspects.clear();
4         passwordAspects.addAll(config.getPasswordAspects());
5         if (passwordAspects.isEmpty()) {
6             setupDefaultPasswordAspects();
7         }
8     }
```

Abbildung 6.5: Violations zu LOD entfernt

Die Abbildung 6.5 zeigt nun das Resultat des Refactorings. In Anhang B.1.1 ist die vollständige Klasse dargestellt.

In Zeile 4 wird auf die Klassenvariable zugegriffen. Dies stellt nun keinen Verstoß zu LOD dar. Ebenso wurden vier Branches entfernt, da die Prüfungen dem Aufrufer dieser Klasse übergeben wurden. Die Verwendung des NullObjectPatterns[5] würde weitere Vorteile bringen.

Der Aufruf

```
engine.getMandantConfiguration().getPasswordAspectConfig();
```

wurde eine Abstraktionsebene höher ausgelagert. So ist nun eine weitere Violation bei dem Aufrufer zu erwarten. Auch an dieser Stelle muss nun ein weiteres Refactoring durchgeführt werden. Hier wurde lediglich eine neue Klassenvariable

[5]siehe http://www.oodesign.com/null-object-pattern.html

```
engine.getMandantConfiguration();
```

initialisiert, da die Klasse MandantConfiguration an mehreren Stellen benutzt wurde.

Law of Demeter und Delegation

Folgend soll ein weiteres Refactoring vorgestellt werden, indem Violations zu LOD entfernt werden. Dabei handelt es sich um ein Refactoring, indem Aufrufe per Delegation weitergeleitet werden.

Die Methode *enableActiveSubForm* beinhaltet insgesamt 6 Violations zu LOD in den Zeilen 18 - 20 (Abbildung 6.6).

```
1   public void enableActiveSubForm() {
2       ManageAccountPermissionsForm parent = view.
            getManageAccountPermissionsForm();
3
4       boolean currentCustomerExistingModeSelected =
            isCurrentCustomerExistingModeSelected(parent);
5       boolean customerAccountsButtonVisibility =
            isGetCustPermissionRequestPermitted()
6               && currentCustomerExistingModeSelected;
7
8       view.setCustomerAccountsButtonVisible(
            customerAccountsButtonVisibility);
9       view.setCustomerAccountsButtonEnable(
            currentCustomerExistingModeSelected);
10
11      view.setCustomerAccountsCloneButtonVisible(
            customerAccountsButtonVisibility);
12      view.setCustomerAccountsCloneButtonEnable(
            currentCustomerExistingModeSelected);
13
14      view.setUserCloneButtonVisible(currentCustomerExistingModeSelected &&
             isGetUserRequestPermitted());
15
16      view.setNextButtonVisible(parent.hasAccounts());
17
18      parent.getSelectedAccountsForm().setEditAllowed(true);
19      parent.getSelectedAccountsForm().setDeleteAllowed(true);
20      parent.getSelectedAccountsForm().setSelectAllowed(true);
21  }
```

Abbildung 6.6: Methode enableActiveSubForm mit 6 Violations zu LOD

Der Test dieser Methode zeigt, dass der Zugriff auf das Objekt *parent* ein zusätzliches Testsetup benötigt, indem ein zusätzliches Mockobjekt erstellt werden muss (Abbildung 6.7 Zeile 23 - 28). Auch muss sichergestellt werden, dass die Methode

getSelectedAccountsForm dreimal aufgerufen wird. Es sind insgesamt also 6 Zeilen in einem Test notwendig, die Zeilen 18 - 20 der Methode *enableActiveSubForm* zu testen.

```
1   @Test
2   public void testEnableActiveSubForm() {
3       EasyMock.expect(view.getManageAccountPermissionsForm()).andReturn(
            parent);
4       EasyMock.expect(parent.getCurrentCustomerMode())
5               .andReturn(ManageAccountPermissionsForm.
                    CURRENT_CUSTOMER_EXISTING);
6
7       User user = new User();
8       user.addTransactionConstraints(new TransactionConstraints("
            GetCustPermissionRequest", 0));
9       user.addTransactionConstraints(new TransactionConstraints("
            GetUserRequest", 0));
10      EasyMock.expect(clientEngine.getUser()).andReturn(user).times(2);
11
12      view.setCustomerAccountsButtonVisible(true);
13      view.setCustomerAccountsButtonEnable(true);
14
15      view.setCustomerAccountsCloneButtonVisible(true);
16      view.setCustomerAccountsCloneButtonEnable(true);
17
18      view.setUserCloneButtonVisible(true);
19
20      EasyMock.expect(parent.hasAccounts()).andReturn(true);
21      view.setNextButtonVisible(true);
22
23      SelectedAccountsForm selectedAccountsForm = EasyMock.createMock(
            SelectedAccountsForm.class);
24      EasyMock.expect(parent.getSelectedAccountsForm()).andReturn(
            selectedAccountsForm).times(3);
25
26      selectedAccountsForm.setEditAllowed(true);
27      selectedAccountsForm.setDeleteAllowed(true);
28      selectedAccountsForm.setSelectAllowed(true);
29
30      EasyMock.replay(view, parent, clientEngine, selectedAccountsForm);
31
32      sut.enableActiveSubForm();
33      EasyMock.verify(view, parent, clientEngine, selectedAccountsForm);
34  }
```

Abbildung 6.7: Testmethode zu enableActiveSubForm

Die Methode *parent.getSelectedAccountsForm* wird immer in einem Kontext verwendet. Dabei wird entschieden, ob das aktuell angezeigte Formular bearbeitet werden darf oder nicht. Hier lässt sich nun dieser Aufruf zusammenfassen, indem das Objekt *parent* eine neue Methode *accountsFormAllAllowed* bekommt. In dieser Methode werden nun die Methoden *setEditAllowed, setDeleteAllowd* und *setSelectedAllowed* aufgerufen.

Die Zeilen 18 - 20 aus Abbildung 6.6 können nun mit

```
parent.accountsFormAllAllowed();
```

ersetzt werden. Dabei fallen nun mehrere Zeilen im Test weg, die in Abbildung 6.8 dargestellt sind. Es muss nun lediglich im Test der Aufruf *parent.accountsFormAllAllowed*, anstelle des Quellcodes der Abbildung 6.8, hinzugefügt werden.

```
1        SelectedAccountsForm selectedAccountsForm = EasyMock.createMock(
             SelectedAccountsForm.class);
2        EasyMock.expect(parent.getSelectedAccountsForm()).andReturn(
             selectedAccountsForm).times(3);
3
4        selectedAccountsForm.setEditAllowed(true);
5        selectedAccountsForm.setDeleteAllowed(true);
6        selectedAccountsForm.setSelectAllowed(true);
```

Abbildung 6.8: Quellcode, der aus dem Test entfernt werden kann

Nun kann noch ein weiteres Refactoring durchgeführt werden. Die Zeile 16 in Abbildung 6.6 zeigt, dass das Objekt *parent* vom Typ ManageAccountPermissionsForm am Objekt *view* vom Typ AccountOverviewForm Verwendung findet. In Zeile 2 der Abbildung 6.6 wird jedoch über das Objekt *view* nach einem Objekt *parent* gefragt. Auch dieser Aufruf stellt eine Violation zu LOD dar. Der Aufruf

```
parent.hasAccounts();
```

kann so in die Methode *setNextButtonVisible* verschoben werden. Hier spart man sich eine weitere Zeile im Test, wenn das Objekt *parent* in der Klasse *AccountOverviewForm* lediglich als Stub bereitgestellt wird.

Die Methode *enableActiveSubForm* ist in Abbildung 6.9 das Ergebnis des Refactorings.

```
1     public void enableActiveSubForm() {
2         boolean currentCustomerExistingModeSelected =
3                 isCurrentCustomerExistingModeSelected(view.
                      getManageAccountPermissionsForm());
4         boolean customerAccountsButtonVisibility =
              isGetCustPermissionRequestPermitted()
5                 && currentCustomerExistingModeSelected;
6
7         view.setCustomerAccountsButtonVisible(
              customerAccountsButtonVisibility);
8         view.setCustomerAccountsButtonEnable(
              currentCustomerExistingModeSelected);
9
10        view.setCustomerAccountsCloneButtonVisible(
              customerAccountsButtonVisibility);
11        view.setCustomerAccountsCloneButtonEnable(
              currentCustomerExistingModeSelected);
12
13        view.setUserCloneButtonVisible(currentCustomerExistingModeSelected &&
               isGetUserRequestPermitted());
14
15        view.setNextButtonVisible();
16        view.accountsFormAllAllowed();
17    }
```

Abbildung 6.9: Methode enableActiveSubForm nach dem Refactoring

Die Verwendung des Objektes *parent* wurde weiter refaktorisiert, indem der Aufruf der Methode *accountsFormAllAllowed* nun ebenfalls zum Typ AccountOverviewView hinzugefügt wurde.

Eine bessere Aufteilung der Verantwortlichkeiten der Klasse *AccountOverviewLogic* brachte zwei schlankere Tests hervor und ebenso ist die Methode *enableActiveSubForm* etwas geschrumpft. Die Klasse *AccountOverviewLogic* enthält weitere Violations zu LOD. Diese können auf ähnliche Weise entfernt werden. Besonders in Verwendung mit Mockobjekten ist eine Vereinfachung der Tests zu erwarten, wenn Violations zu LOD generiert werden.

False-Postives

Für diese Fallstudie wurden alle möglichen False-Postives der Metrik VOD analysiert. Dabei sind einige Violations aufgefallen, die im Fokus der Testbarkeit keine Violations darstellen.

Folgende Bereiche konnten dafür eingegrenzt werden:

- Bereitstellung eins Logging-Mechanismus

- Zugriffe auf Entity-Klassen[6]
- Statische Instanziierungen (KLasse.newInstance())
- Lokale Instanziierungen, die in anderen Methoden durchgeführt werden
- Factory-Klassen
- Trennung von Deklaration und Initialisierung (Fehler der LOD-Implementierung)

Der Logging-Mechanismus sollte aus der Wertung für VOD entfernt werden. Hierzu muss bei der Prüfung eines Verstoßes entsprechend reagiert werden.

Die Zugriffe auf Entity-Klassen sind ebenfalls testunkritisch. Eine Klasse Customer, die einen weiteren Datentyp Account enthält, können Aufrufe wie

```
Account account = customer.getAccount();
account.getAccountNr();
```

durchaus ausgeführt werden. Wichtig in diesen Zusammenhang ist nur, das jegliche Entity-Klassen keine Logik enthalten.

Auch müssten zu den False-Positives generierte Violations, die auf fremde Bibliotheken beruhen, hinzugefügt werden. An diesen Stellen lässt sich ohne Weiteres nichts ändern. Mit dem gezeigten Beispiel 6.3.2 und der Verringerung des Testaufwandes ist nicht in jedem Fall zu rechnen.

Für die Metrik VOD ist eine Langzeitstudie sinnvoll. Erst durch die Integration in den Entwicklungsalltag können weitere False-Positives bzw. Positive-False der Metrik VOD aufgedeckt werden.

Die Abbildung 6.10 zeigt alle generierten Violations der untersuchten Projekte. Hinzugenommen wurden ebenfalls Falschaussagen und weitere Quellcodeabschnitte, die von der Metrik VOD unentdeckt blieben.

Fazit

Zusammengefasst kann gesagt werden, dass die Metrik VOD noch nicht produktionsreif ist. Die Aussagen der Metrik sind noch zu unpräzise und würden bei den Entwicklern wenig Akzeptanz finden. An der Metrik VOD müssen dementsprechend weitere Änderungen vorgenommen werden.

[6]Klassen, die nur der Datenhaltung dienen (keine Logik enthalten)

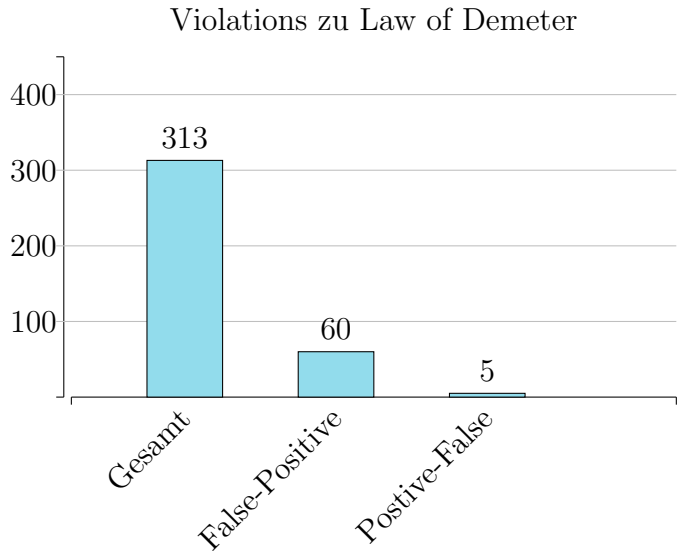

Abbildung 6.10: Violations zu Law of Demeter

6.3.3 Die Reduktionsmetrik rACDh

Die Reduktionsmetrik rACDh soll folgend ebenfalls untersucht werden. Auch hier war das Ziel, bei erkannten testkritischen Abhängigkeiten ein Refactoring durchzuführen. Ebenfalls standen andere Metriken des Modells unter Beobachtung.

Grenzen der Metrik rACDh

Der Einsatz der Metrik zeigte in dieser Fallstudie klare Grenzen. So versagt die Metrik, wenn eine Menge von neuen Klassen einem Projekt hinzugefügt werden. Bewegt sich eine Violation genau an einer konfigurierten Grenze (beispielsweise 10%), so wird mit einer größeren Anzahl von neuen Klassen, diese aufgedeckte testkritische Abhängigkeit dann nicht mehr als testkritisch eingestuft. Mit steigender Klassenanzahl verringern sich also die testkritischen Abhängigkeiten. Das ist ein klarer Nachteil der Reduktionsmetrik rACDh.

Gegenüberstellung der Differenzen der Metrik ACDh zu Projektumfang

Folgend sollen Messungen im Bereich der Steigerung des ACDh-Wertes von 5% bis 25% durchgeführt werden. Dabei werden alle Violations untersucht und gegebenenfalls als False-Positive gewertet. Untersucht werden sollen Projekte im Bereich von 20 bis 1000 Klassen.

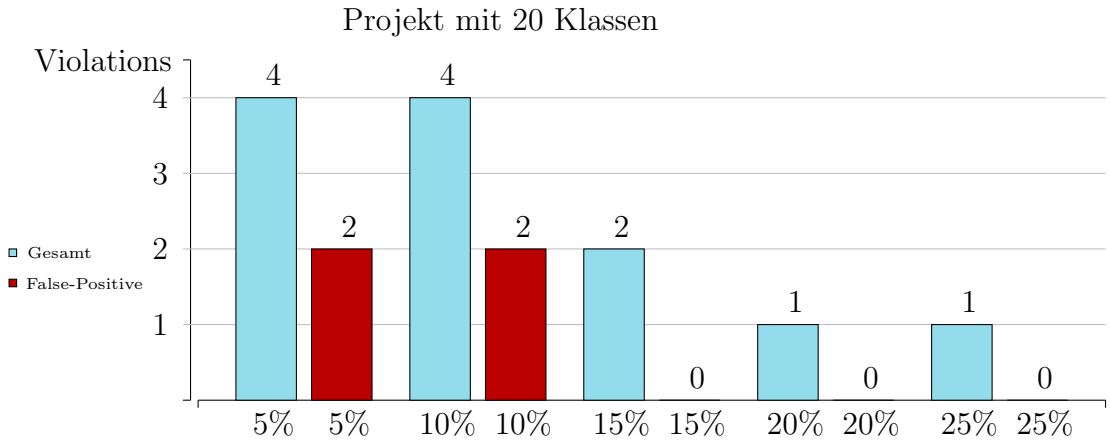

Abbildung 6.11: Reduktionsmetrik mit 20 Klassen

Das Säulendiagramm aus Abbildung 6.11 zeigt alle gefundenen Abhängigkeiten (Violations zur Reduktionsmetrik rACDh). Ebenfalls wurden dabei Violations, die keine testkritischen Abhängigkeiten sind, als False-Positive markiert. Bei einer geringen Differenz wurden False-Positives gefunden. Ab einer Differenz von 15% der Metrik ACDh traten keine Falschaussagen mehr zu[7].

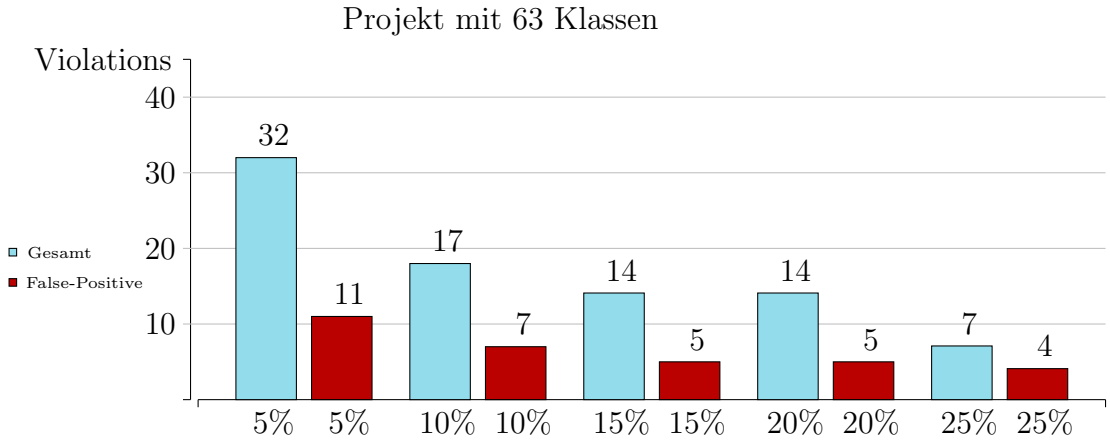

Abbildung 6.12: Reduktionsmetrik mit 63 Klassen

Das Projekt mit einem Umfang von 63 Klassen (Abbildung 6.12) zeigte schlechtere Werte wie das Projekt mit 20 Klassen Umfang. Mit steigender Differenz der Metrik ACDh wurden korrekt eingestufte Abhängigkeiten unterdrückt. Auch mit hoch eingestellten Werten der Differenz zur Metrik ACDh wurden dennoch Abhängigkeiten als Falschaussagen eingestuft. Erwartet wurde, je höher der Wert, desto weniger Falschaussagen generiert werden. So wurde erwartet, dass mit Werten größer 20% ACDh-Differenz keine Falschaussagen mehr generiert werden.

[7]Nach Positive-False Abhängigkeiten wurde aus Zeitgründen nicht gesucht

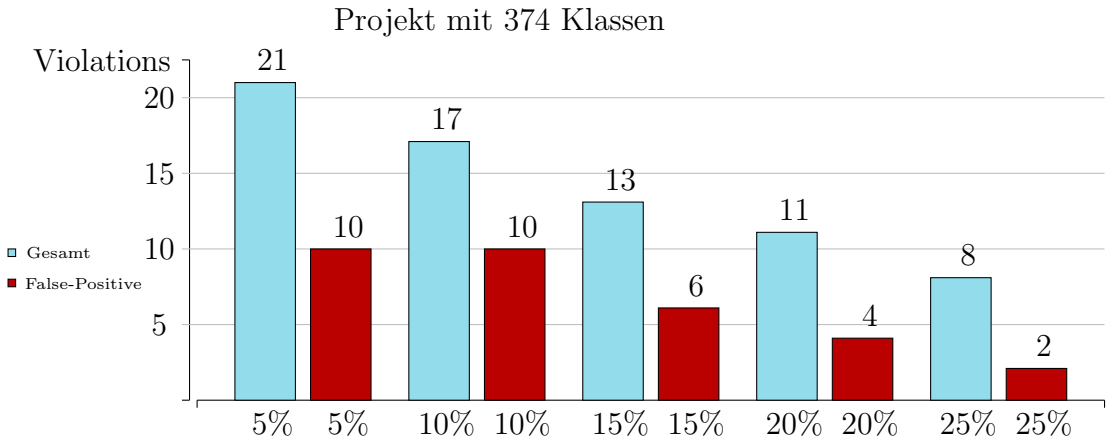

Abbildung 6.13: Reduktionsmetrik mit 374 Klassen

Bei dem Projekt mit einem Umfang von 374 Klassen (Abbildung 6.13) sehen die Ergebnisse ähnlich des Projektes aus Abbildung 6.12 aus. Auch hier werden testkritische Abhängigkeiten unterdrückt und bei hohen Werten sind ebenfalls Falschaussagen zu finden.

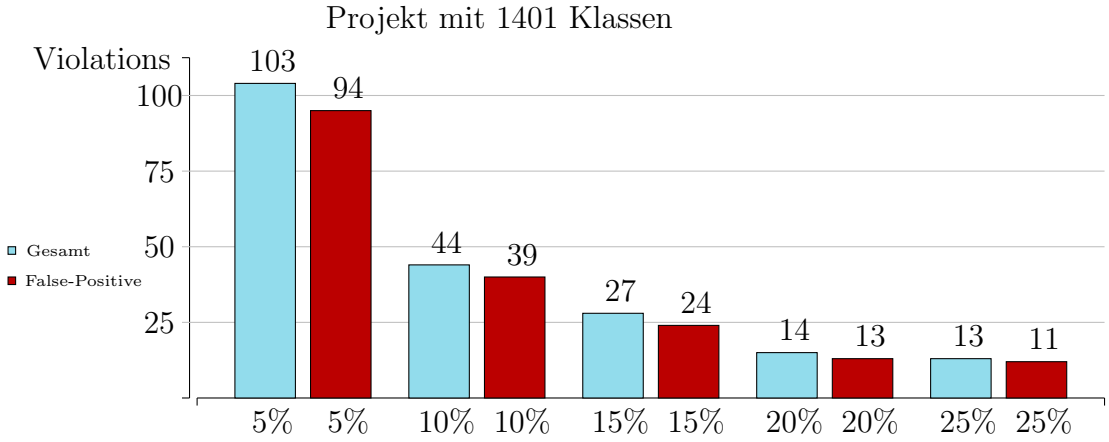

Abbildung 6.14: Reduktionsmetrik mit 1401 Klassen

Das Projekt mit einem Umfang von 1401 Klassen zeigte sehr schlechte Werte der Reduktionsmetrik rACDh. So werden, egal mit welcher Differenz gearbeitet wird, zum größten Teil Falschaussagen generiert. Die Ergebnisse der Metrik sind für dieses Projekt nutzlos.

Refactoring von testkritischen Abhängigkeiten

Eine testkritische Abhängigkeit kann entfernt werden, indem die Instanziierung aus der entsprechenden Klasse entfernt wird. Entweder muss diese Abhängigkeit dem Konstruktor

der Klasse übergeben werden oder in der entsprechenden Methode als Parameter eingereicht werden. In beiden Fällen wird diese testkritische Abhängigkeit dem Aufrufer der Klasse überlassen. Auch hier wird diese Abhängigkeit ebenfalls als testkritisch eingestuft. So wird die Instanziierung der Klasse in Richtung des Programmeinstieges verschoben. Das Problem der Instanziierung wird somit lediglich einer anderen Klasse überlassen.

Der Einsatz eines Dependency Injection Frameworks kann hier jedoch Abhilfe schaffen. Ein DI-Framework kann beispielsweise dafür eingesetzt werden, um testkritische Abhängigkeiten zu erzeugen. So ist ein Entwickler nicht dazu gezwungen, entsprechende Instanzen selber zu erzeugen und kann sich auf die Implementierung der Geschäftsprozesse besser konzentrieren.

Fazit

Zusammengefasst kann gesagt werden, dass die Metrik rACDh sehr unzuverlässig arbeitet. Zum einen ist die Reduktionsmetrik in großen Projekten zu unpräzise, zum anderen können in kleineren Projekten testkritische Abhängigkeiten, mit einem Hinzufügen neuer Klassen, entfernt werden. Aus diesen Gründen muss über die Entfernung der Reduktionsmetrik aus dem Modell nachgedacht werden. Auch eine zusätzliche Absicherung mit der Metrik CDh brachte keinen nennenswerten Fortschritt.

Folgende Punkte sind in dieser Fallstudie aufgetaucht, die die Reduktionsmetrik unberücksichtigt lässt:

- Ignorierung von Entity-Klassen
- Ignorierung von Factory-Klassen (den darin enthaltenen Instanziierungen)
- Tiefe des Abhängigkeitsgraphen
- Funktionalitätsumfang, auf den zugegriffen wird

Eine Betrachtung der Tiefe des Abhängigkeitsgrapen sowie eine parallele Betrachtung des gesamt benutzten Funktionalitätsumfanges innerhalb dieses Abhängigkeitsgraphen war bei der Entscheidung für ein False-Positive am hilfreichsten. So müsste eine Metrik, die diese Kriterien berücksichtigt, parallel ausgewählt werden. Dabei können beide Metriken verglichen werden. Hierfür konnte bis zur Abgabe dieser Master-Thesis keine entsprechende Metrik gefunden werden.

6.3.4 Die McCabe-Metrik

Folgend soll die McCabe-Metrik betrachtet werden. Traten hohe Werte auf, so war den Entwicklern klar, das solch eine Klasse nur schwer testbar ist bzw. ein Vielzahl an Tests benötigt.

Anzahl der Testfälle

Tabelle 6.2 zeigt zwei Projekte, die per TDD entwickelt wurden. Beide Projekte enthalten maximal eine Klassenkomplexität von 20. Beide Projekte wurden mit Unittests zu 100% abgesichert. Auch lassen sich, laut der Entwickler, alle Klassen überschaubar mit Unittests absichern.

Komplexität	Testfälle	Quotient	Klassenkomplexität	Klassen
631	342	1,85	2,9	217
475	311	1,53	5,7	84

Tabelle 6.2: Komplexität und Anzahl der Testfälle auf Systemebene

Aus der Tabelle 6.2 ist ebenfalls gut zu erkennen, das die Klassenkomplexität und die Anzahl der Klassen negativ korrelieren. Je geringer die maximal erlaubte Klassenkomplexität gesetzt wird, desto mehr Klassen sind zu erwarten.

Durchführung eines Refactorings

Für ein Refactoring wurde zunächst nach doppeltem Quellcode gesucht. Bestand dieser, wurden die entsprechenden Quellcodezeilen in eine neue Klasse ausgelagert. Entsprechend wurden neue Instanzen erzeugt, wodurch die Metrik rACDh in manchen Fällen eine Violation generierte. Durch dieses Refactoring wurden die Komplexitätswerte einer Klasse verringert. Ein nächster Schritt war es, per *Extract Method* innerhalb einer Klasse, Funktionalität in einzelne Methoden auszulagern. Hierbei bekommt man einen sehr guten Überblick der Klasse und welche Funktionalitäten in eine neue Klasse ausgelagert werden können. Jedoch sind sehr hohe Komplexitätswerte (> 100) mit hohem Aufwand verbunden. Zudem ist eine hohe Anzahl von neuen Klassen zu erwarten, die ebenfalls wieder instanziiert werden müssen.

Fazit

Die Komplexität einer Klasse zeigt recht zuverlässig den zu erwartenden Testaufwand bzw. ab wann eine Klasse zu viele Aufgaben erledigt. Eine exakte Grenze konnte jedoch nicht ermittelt werden. Dies wird sich in einer Langzeitstudie zeigen.

6.3.5 Die Metrik LCOM4

Es soll nun die Metrik LCOM4 genauer untersucht werden.

Klassen mit geringem Umfang

Die Klasse ReplyToInitialRequestForPublicKeys hat einen Umfang von 18 LOC.

```
1 public class ReplyToInitialRequestForPublicKeys extends
      GenericFinancialInstitutionMessage {
2
3     public List<PublicKeyTransmission> getPublicKeyTransmissions() {
4         return getAll(PublicKeyTransmission.class);
5     }
6
7     public BankParameterData getBankParameterData() {
8         BankParameterData bankParameterData = new BankParameterData();
9         for (Segment segment : getDataSegments()) {
10            bankParameterData.addSegment(segment);
11        }
12        return bankParameterData;
13    }
14
15 }
```

Abbildung 6.15: Negate Conditionals Mutator

Aus der Abbildung 6.15 lässt sich zunächst feststellen, dass diese Klasse einen eher geringen Umfang hat. Diese Klasse lässt sich zudem mit drei Unittests zu 100% absichern.

Es stellt sich nun zurecht die Frage, ob diese Klasse als testkritisch angesehen werden kann. Da man eher von einem Designproblem sprechen kann, spielt für diese Klasse die Testbarkeit eher eine untergeordnete Rolle.

Mit Rücksprache des verantwortlichen Entwicklers, wird diese LCOM4-Metrik dennoch nicht als *False-Positive* gesetzt, da auch hier gesagt wurde, dass ein Designproblem vorliege. Dabei wurden folgende Punkte angesprochen:

- Es wurde nicht genügend über die Umsetzung nachgedacht

- Der Test-First Ansatz wurde nicht zu 100% durchgeführt

Es kann also zum einen der Umfang einer Klasse mit dieser Metrik begrenzt werden, zum anderen werden Entwickler auf vorhandene Designprobleme hingewiesen (SRP).

Im durchgeführten Refactoring (4.4.6 Refactoring von Tristan - End of Day processing) wurde gezeigt, dass doppelter Quellcode und ebenfalls der Testumfang eingegrenzt werden konnte. Es wurde ein Interface bereitgestellt, dass einen dazu gezwungen hat, entsprechende Methoden zu implementieren. Dadurch wurden LCOM4-Werte und Duplications in die Höhe getrieben. Auch waren Klassen mit wenig Umfang vertreteten, doch es konnte gezeigt werden, dass durch ein entsprechendes Design, weniger Tests notwendig waren.

Wenn LCOM4-Werte im Zusammenhang mit Vererbung oder Polymorphismus entstehen, so kann durch weiteres Hinzufügen von Funktionalitäten später doppelter Quellcode erzeugt werden.

Die Abbildung 6.16 zeigt die Zugriffspfade der Klasse ChangeKey.

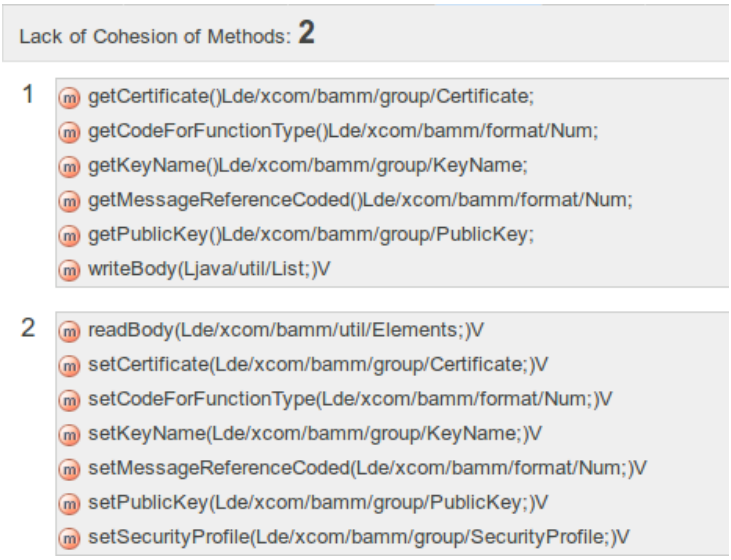

Abbildung 6.16: LCOM4 Darstellung der Klasse ChangeKey

Man erkennt, dass diese Klasse zwei Funktionalitäten anbietet. Einen Pfad mit *writeBody* und den dazugehörigen Getter-Methoden. Daneben einen Pfad mit *readBody* und den dazugehörigen Setter-Methoden. In dieser Klasse werden Server- und Clientfunktionen vereint.

Im Anhang B.2 Die Metrik LCOM4 ist die Klasse ChangeKey2 dargestellt. Auch diese Klasse, mit einem Umfang von 61 LOC, kann mit wenigen Tests zu 100% abgesichert werden.

Betrachtet man jedoch weitere Klassen, die von der Klasse ChangeKey erben, so sind in diesen Klassen ebenfalls gleiche Strukturen zu erkennen, wie in Klasse ChangeKey2. Je mehr Klassen nun implementiert werden (also von ChangeKey erben), desto höher wird die Wahrscheinlichkeit, Duplications zu implementieren.

Die Klasse ChangeKey3 ist im Anhang B.2 zu finden und zeigt dabei gleiches Verhalten wie die Klasse ChangeKey2.

Auch bilden die restlichen Klassen ähnliche Strukturen und können potentiell doppelten Quellcode erzeugen.

Fazit

Die Metrik LCOM4 muss nicht zwingend auf testkritische Klassen zeigen. Im Zusammenhang mit Umfang, Vererbung und Polymorphismus zeigt diese Metrik zum einen Designprobleme, die später zu doppelten Quellcode führen können, zum anderen eine Klasse, die in weitere Klassen aufgeteilt werden kann und dadurch die Testklassen ebenfalls in ihrem Umfang begrenzt werden können.

6.3.6 Die Metrik Survived Mutant

Es soll nun die Metrik Survived Mutant näher untersucht werden. Dabei werden Klassen herangezogen, die eine hohe Anzahl der Violation *Survived Mutant* haben.

Beispiel 1 - Die Klasse InitialRequestForPublicKeys

Die Klasse InitialRequestForPublicKeys beinhaltet 16 Violations der Metrik *Survived Mutant*. Alle Violations beziehen sich dabei auf den *Void Method Call Mutator*[8].

```
1     private Identification2 createIdentification(final Fid fid) {
2         Identification2 identification2 = new Identification2();
3         identification2.setFinancialInstitutionId(fid);
4         identification2.setCustomerId("9999999999");
5         identification2.setCustomerSystemId("0");
6         identification2.setCustomerSystemStatus(0);
7         return identification2;
8     }
```

[8] siehe http://pitest.org/quickstart/mutators/#void-method-call-mutator-void_method_calls

Abbildung 6.17: Void Method Call Mutator

Für die Methode *createIdentification* werden vier Violations generiert (Zeile 3 - 6). Der Unittest der Klasse InitialRequestForPublicKeys prüft also nicht alle zurück gelieferten Informationen der Klasse. So könnte man unbemerkt die CustomerId ändern, ohne das ein Unittest fehlschlagen würde. Die entsprechenden Unittests müssen also erweitert werden, indem weitere Assertions hinzugefügt werden müssen.

Das Beseitigen dieser Violations sollte also kein Problem darstellen und kann ohne erheblichen Aufwand durchgeführt werden. Da die Zeilen 4 - 6 als nicht relevant interpretiert werden können, stellt sich die Frage ob Assertions für diese Zeilen hinzugefügt werden müssen. Wird später ein Refactoring durchgeführt oder entsprechende Parameter abgeändert, können unbemerkt Fehler implementiert werden. Es kann nicht davon ausgegangen werden, dass jeder Entwickler mit Unittests arbeitet. Daher ist es sinnvoll weitere Assertions hinzuzufügen.

Da für diese Klasse eine größere Anzahl von Assertions zu erwarten ist, kann man hierfür neue Assertions-Klassen implementieren. Für die Methode *createIdentification* kann eine neue Klasse AssertIdentifaction2 erstellt werden, die als Parameter die Klasse Identification2 entgegennimmt. So kann diese Klasse auch in anderen Testklassen verwendet werden, ebenso sind weniger Assertions notwendig.

Beispiel 2 - Die Klasse Connection

Die Klasse Connection beinhaltet 12 Violations der Metrik *Survived Mutant*. 11 Violations beziehen sich wiederrum auf den *Void Method Call Mutator* und eine Violation auf den *Negate Conditionals Mutator*[9].

Dieser *Negate Conditionals Mutator* soll nun näher untersucht werden.

```
1         for (Segment segment : orderReply.getSegments()) {
2             Num referenceSegment = segment.getReferenceSegment();
3             if (referenceSegment != null && referenceSegment.asInt() ==
                  reference) {
4                 answers.add(segment);
5             }
6         }
```

Abbildung 6.18: Negate Conditionals Mutator

[9] siehe http://pitest.org/quickstart/mutators/#negate-conditionals-mutator-negate_conditionals

In Zeile 3 wird eine Violation des genannten Typs erzeugt. Aus dieser Bedingung wird temporär die Bedingung

```
if (referenceSegment == null && referenceSegment.asInt() != reference)
```

erzeugt.

Im dafür verantwortlichen Test wird jedoch nur die Anzahl der Segmente überprüft. Wäre dem nicht so, so würde auch für die Zeile 4 eine Violation generiert werden.

Damit der Test für diese Mutation fehlschlägt, muss auch hier eine weitere Assertion eingefügt werden. Hierfür ist lediglich das Prüfen der Segmentnummer erforderlich.

Fazit

Die Metrik *Survived Mutant* zeigt sehr genau, welche Unittests noch Schwächen enthalten. Im Bezug auf die Erweiterbarkeit einer Klasse können hierdurch Probleme entstehen, die auch in einem späteren Verlauf unbemerkt bleiben könnten. So zeigt diese Metrik dem Entwickler, welche Stellen noch besser getestet werden müssen.

6.3.7 Die Metrik Duplicated Blocks

Eine Betrachtung der Metrik Duplicated Blocks soll an dieser Stelle nicht durchgeführt werden. Es wurde bereits bei der Auswahl der Metrik ein Refactoring durchgeführt (siehe 4.4.3 Doppelter Quellcode). Dabei wurde festgestellt, das doppelter Quellcode ebenfalls zu einem Mehraufwand in der Testimplementierung führt.

6.3.8 Die Metrik Switch-Density

Folgend soll die Metrik Switch-Density untersucht werden.

Generell hat sich gezeigt, dass Switch-Statements eher weniger verwendet werden, jedoch existierende Statements durch ein Refactoring hätten entfernt werden können. Hierbei wurden Stichproben entnommen und ein Refactoring durchgeführt. Auch wurden die Refactorings allgemein als etwas komplexer und umfangreicher angesehen, als andere durchgeführte Refactorings.

Refactoring von Switch-Statements

Die Abbildung 6.19 zeigt ein Switch-Statement, welches einem Refactoring unterzogen wurde. Für jeden Branch wurde eine neue Klasse erstellt. Daraus entstanden die Klassen MT353FormatProcessor, MT571FormatProessor und UnknownFormatProcessor.

```
1             switch (type) {
2                 case MT535:
3                     return getMT535(initiatorAccount, assetOverviewResult.
                          getBusinessResult());
4                 case MT571:
5                     return getMT571(initiatorAccount, assetOverviewResult.
                          getBusinessResult());
6                 default:
7                     throw new Exception("unexpected format[" + type + "]");
8             }
```

Abbildung 6.19: Switch-Statement

Für die Instanziierung der Klassen wurde eine Factory erzeugt. Existiert für ein übergebenes Format kein Processor, so wird der UnknownFormatProcessor instanziiert. Zuvor wurde eine Exception geworfen und in selbiger Klasse gefangen. Im Catch-Block wurde ein Result erstellt und zurückgegeben. Dies übernimmt nun die Klasse UnknownFormatProcessor. Ebenso wurde ein Interface FormatProcessor erstellt, dass alle genannten Klassen implementiert.

```
1 public class MTFormatFactory {
2
3     private Map<PortfListRequest.MTType, FormatProcessor> processors
4             = new HashMap<PortfListRequest.MTType, FormatProcessor>();
5
6     public MTFormatFactory(final TimeProvider timeProvider) {
7         processors.put(PortfListRequest.MTType.MT535, new MT535Processor(
              timeProvider));
8         processors.put(PortfListRequest.MTType.MT571, new MT571Processor(
              timeProvider));
9     }
10
11    public FormatProcessor getFormatProcessor(final PortfListRequest.MTType
          type) {
12        if (null != processors.get(type)) {
13            return processors.get(type);
14        }
15        return new UnknownFormatProcessor(type);
16    }
17 }
```

Abbildung 6.20: Klasse MTFormatFactory

Die Instanziierung der MTFormatFactory wurde dann mit dem Switch-Statement ersetzt (Abbildung 6.21). An dieser Stelle wurde eine Violation der Metrik VOD und der Metrik rACDh generiert. Beide Violations sind jedoch Falschaussagen, da hier wieder auf eine Factory-Klasse zugegriffen wird. Ebenso erfolgt der Aufruf der Methode *process* über ein Interface. Der entsprechende Typ ist somit erst zur Laufzeit bekannt. Die rACDh-Metrik lässt dies unberücksichtigt, da ein solcher Aufruf mit einem Mockobjekt oder Stubobjekt[10] simuliert werden kann. Hier kann also mit einem entspechendem Testsetup kein tiefer Abhängigkeitsgraph entstehen. Auch wird dabei auf keine Funktionalität zugegriffen.

```
1            MTFormatFactory mtFormatFactory = new MTFormatFactory(
                 timeProvider);
2            return mtFormatFactory.getFormatProcessor(type).process(
3                    initiatorAccount,
4                    assetOverviewResult.getBusinessResult());
```

Abbildung 6.21: Switch-Statement mit MTFormatFactory ersetzt

Die ursprüngliche Klasse, die alle Aufgaben übernommen hat, ist dadurch in ihrem Umfang stark reduziert wurden. Dafür mussten jedoch sechs neue Klassen erstellt werden.

Die Tabellen 6.3 und 6.4 zeigen eine Übersicht des Refactorings und den draus resultierenden Komplexitäten. Dabei lag das Coverage der Unittest bei 68%.

Klasse	CCN2	Testfälle
PortfolioList	64	9

Tabelle 6.3: Die Klasse PortfolioList vor dem Refactoring

Klasse	CCN2	Testfälle
PortfolioList	18	1
MTFormatFactory	4	0
MT535Processor	12	1
MT571Processor	11	1
InstrumentDescription	7	5
SecurityAccount	14	1
UnknownFormatProcessor	2	0
Gesamt	68	9

Tabelle 6.4: Bestehende und neue Klassen nach dem Refactoring

[10] Stellvertreter

Bei diesem Refactoring wurden keine neuen Tests implementiert. Zwar wurde die Komplexität etwas erhöht (Konstruktoren einer Klasse), jedoch lassen sich die einzelnen Klassen nun besser testen. Hier zeigt sich ebenfalls, dass die eigentliche Kernfunktionalität (MT535Processor, MT571Processor) der Klasse PortfolioList mit lediglich zwei Unittests abgesichert ist.

Fazit

Die Metrik Swicht-Density hilft dabei Klassen zu finden, die in weitere Klassen aufgeteilt werden können. Das Resultat sind kleinere Klassen, die sich besser testen lassen.

6.4 Zusammenfassung

Da diese Fallstudie in einem kleinen Rahmen durchgeführt wurde, wurden keine Reviews geplant. Es sollte zunächst untersucht werden, wie genau die ausgewählten Metriken arbeiten. Dabei hat sich gezeigt, dass die Metrik *rACDh* und die Metrik *VOD* noch unzuverlässig arbeiten. Erst wenn das Modell zuverlässige Ergebnisse liefert, können Reviews angeknüpft werden. Die zusätzlich angegebenen Aufwandseinschätzungen zum Entfernen einer Violation konnten mit dieser Fallstudie nicht überprüft werden.

Ebenso hat sich generell gezeigt, dass durch die ausgewählten Metriken mehr Klassen erzeugt werden. Entweder verrichten Klassen zu viel Arbeit oder die Verantwortlichkeiten einer Klasse sind nicht korrekt aufgelöst. Unter Berücksichtigung dieser Aspekte werden neue Klassen entstehen, die instanziiert werden müssen, sodass ebenfalls weitere testkritische Abhängigkeiten entstehen können. Dies zeigt, dass der Einsatz eines Dependency Injection Frameworks ein nächster wichtiger Schritt in Richtung Testbarkeit einer Klasse sein kann.

Durch den Abbruch des Build-Prozesses haben sich ebenfalls die Werte der Metriken verbessert. Daher soll dieser Ansatz beibehalten werden. Lediglich eine Verkürzung der Analysezeiten von Sonar ist erforderlich.

Kapitel 7

Zusammenfassung und Ausblick

Im letzten Kapitel werden die wichtigsten Punkte zusammengefasst dargestellt. Ebenso soll ein Ausblick auf zukünftige bzw. anknüpfende Arbeiten gegeben werden.

7.1 Zusammenfassung

Ziel dieser Arbeit war es, Metriken in den Softwareentwicklungsprozess zu integrieren und damit ein dauerhaftes Messverfahren einzuführen. Es wurde eine Möglichkeit gezeigt, wie Metriken in einen Softwareentwicklungsprozess integriert werden können. Hierfür wurde ein individuelles Qualitätsmodell aufgestellt, dass versucht, die entsprechend festgelegten Ziele in dieser Abteilung zu unterstützen. Dabei wurden Umfragen und Interviews durchgeführt, um diese Ziele genauer definieren zu können. Aus den Umfragen, Interviews und einem Meeting hatte sich ein Ziel dieser Abteilung herauskristalisiert. Es sollte die Erweiterbarkeit einer Klasse gemessen werden. Dieses Ziel baute hauptsächlich auf der Testbarkeit einer Klasse auf. Das Coverage von Unittests und das Coverage von Integrationstests sind im Modell Erweiterbarkeit lediglich Entscheidungsträger.

Mit diesem Wissen, dass die Testbarkeit einer Klasse gemessen werden soll, wurde auf den Einsatz von GQM verzichtet. In dieser Abteilung wird bereits mit Unittests gearbeitet wodurch die Entwickler direkt befragt werden konnten, indem sie Quellcode zeigen sollten, der nach der Meinung der Entwickler schwer testbar ist.

Daraus ergaben sich folgende Metriken, die in das Qualitätsmodell aufgenommen wurden:

- Violation of Demeter
- LCOM4
- McCabe
- Switch-Density
- Survived Mutant
- Reduktionsmetrik rACDh (CDh, ACDh)
- Komplexität von Duplicated Blocks

Diese Metriken wurden anschließend als Plugin für Sonar implementiert, da es sich um Produktmetriken[1] handelt (Erweiterbarkeit einer Klasse). Sonar wurde eingesetzt, da es einerseits das Messen in Verbindung mit einem CI-Server vereinfacht und ebenfalls eine Bereitstellung der Messergebnisse zentral über ein Webinterface anbietet.

Für eine bessere Integration des Einsatzes von Sonar wurde mittels des CI-Servers Jenkins und dem Build-Pipeline-Plugin für Jenkins die Sonaranalyse in den Entwicklungsprozess fest integriert. Hierfür wurde das Plugin erweitert, indem es Metriken *beobachten* kann und bei Feststellung einer Verschlechterung einer Metrik den Build-Prozess als Fehlschlag markiert. Daraus ergibt sich, dass beispielsweise keine Integrationstests oder auch kein Release mehr durchgeführt werden kann, solange die Metrik einen schlechteren Wert aufweist. Dies gilt auch für hinzugefügte Violations.

Im letzten Abschnitt wurde eine Fallstudie durchgeführt, die jedoch in einem kleinen Umfeld stattgefunden hat. Es wurden bewusst die Entwickler bei der Auswahl der Metriken mit einbezogen, indem diese beispielsweise die Stellen aufzeigen konnten, die für sie als schlecht testbar galten. Trotz der Mitwirkung der Entwickler ergab sich eine recht negative Resonanz, für einen ersten produktiven Einsatz der Metriken. In Verbindung mit einer umfangreicheren Fallstudie, hätten sich dadurch noch weitere positive als auch negative Aspekte finden können.

Jedoch zeigte auch bereits die durchgeführte Fallstudie in einem kleinen Rahmen, dass nicht alle Metriken das zeigen, was eigentlich erwartet wurde. So ist die Metrik VOD noch verbesserungswürdig. Ebenfalls zeigte sich das die Metrik rACDh eine größere Menge an Falschaussagen generiert. Da bei der Feststellung einer Falschaussage einer Violation ebenfalls die Entwickler mitentscheiden konnten, kann nicht sichergestellt werden, dass auch wirklich alle Einstufungen zutreffend sein müssen. Im Zusammenhang mit Reviews,

[1] statische Quellcodeanalyse

in denen einzelne Metriken in einem Team beispielsweise betrachtet werden können, sind bessere bzw. genauere Ergebnisse zu erwarten.

Zusammengefasst kann gesagt werden, dass lediglich die Spitze des Eisberges betrachtet wurde. Soll der Einsatz von Metriken erfolgreich fortgeführt und verstärkt werden, ist weiterer enormer Aufwand notwendig. Auch muss dementsprechend verantwortliches Personal gefunden werden, dass sich mit diesem Thema auseinandersetzt.

7.2 Ausblick

Das aufgestellte Modell darf nicht als vollständig betrachtet werden. Die Ergebnisse der Metrik rACDh sind als fragwürdig eingestuft wurden. Jedoch sollen vorerst Messungen von testkritischen Abhängigkeiten weiterhin mit der Metrik rACDh fortgeführt werden. Parallel dazu kann eine weitere Metrik mit den angesprochenen Kriterien implementiert werden. So lassen sich beide Metriken vergleichen und eventuell Vorteile und Nachteile beider Metriken gegenüberstellen. Ebenfalls kann nach weiteren Metriken gesucht werden, die die Testbarkeit einer Klasse versuchen zu beurteilen bzw. die dies indirekt messen.

Ein weiterer wichtiger Punkt und gleichzeitig ebenfalls entscheidend für einen produktiven Einsatz von Sonar und den darin enthaltenen Metriken, ist eine Beschleunigung der Analysezeiten. Bisweilen wird von Sonar hierfür keine Möglichkeit angeboten. Es existiert jedoch ein Plugin, dass Messungen am Quellcode vornimmt, der kürzlich bearbeitet wurden ist. Dabei bleiben jedoch alle anderen Dateien, die nicht in diese Zeitspanne fallen, unberücksichtigt. Somit bleiben diese in den Messergebnissen ebenfalls unberücksichtigt. Weiterhin kommen hierfür nur Metriken in Frage, die auf Methoden- bzw. Klassenebene arbeiten. Metriken auf Systemebene müssen dafür ausgeschlossen werden (beispielsweise die Reduktionsmetrik rACDh). Das bestehende Plugin lässt sich jedoch daraufhin erweitern, indem es zusätzlich mit einem Versionsverwaltungssystem zusammenarbeitet und entsprechend nur die Dateien untersucht werden, die einer Veränderung unterzogen wurden oder neu hinzugekommen sind. Was noch nicht klar ist, ob entsprechende Metriken auf Systemebene davon zusätzlich ausgeschlossen werden können.

Weiterhin wurde mit dem Plugin ein CSV-Export implementiert, der eine spätere externe Validierung ermöglichen soll. Eine externe Validierung bedarf einer empirischen Untersuchung, ob die Metriken tatsächlich im Zusammenhang mit dem Qualitätsmerkmal Testbarkeit stehen. Die Relation zwischen einer internen und externen Variable muss eine stetig ansteigende Funktion ergeben. Kleine Änderungen der internen Variable sollten auch nur kleine Änderungen an der externen Variable hervorrufen. D.h. ändert sich

beispielsweise die Komplexität einer Klasse nur geringfügig, so sollte sich ebenfalls der Testaufwand und die Testbarkeit nur geringfügig ändern. Auch können, speziell mit dem Einsatz von Sonar, Violations zusammengefasst und bewertet werden. Daraus resultierend lassen sich einzelne Metriken ebenfalls extern validieren.

Werden testkritische Abhängigkeiten richtig erkannt und können ausgelagert werden, so ist der Einsatz eines Dependency Injection Frameworks ein weiterer wichtiger Punkt, die Testbarkeit einer Klasse bzw. eines Projektes zu verbessern. Der Einsatz eines entsprechenden DI-Frameworks wurde jedoch in dieser Arbeit nicht besprochen, da es den Rahmen dieser Arbeit sprengen würde. Besitzt eine Metrik Potential, testkritische Abhängigkeiten sicher zu entdecken, so kann parallel ein DI-Framework eingesetzt werden und die Instanziierung entsprechender testkritischer Abhängigkeiten von einem Framework übernommen werden.

Auch ist die Durchführung von Reviews in Zukunft ein wichtiger Punkt. Wenn mit Metriken gearbeitet werden soll, kann mit Hilfe von Reviews der Einsatz von Metriken weiter verstärkt werden. Ebenfalls können Reviews dabei helfen Metriken besser zu verstehen, was letzten Endes der Anwendung und Interpretierung der einzelnen Metriken zu Gute kommen kann.

In dieser Arbeit wurden nur Produktmetriken untersucht bzw. eingesetzt, die auf der statischen Analyse von Quellcode beruhen. Auch der Einsatz weiterer Metriken (auch anderer Kategorien) soll in Zukunft untersucht werden. So wurde bereits darüber diskutiert, Tickets, die im Ticketsystem von einem Kunden eingereicht werden, zu zählen. Dabei soll entschieden werden ob die eingestellten Tickets mangelhaft sind oder nicht. Hier lässt sich entsprechend über einen Zeitraum eine Statistik abrufen, die zeigen soll, wie versiert der Kunde mit dem Produkt und wie es um sein Fachwissen bestellt ist. Auch andere Interpretationen sind möglich.

Literaturverzeichnis

[2as07] 2ask.de. *Leitfaden statistische Auswertung*. 2ask.de, http://http://www.2ask.de, 2007. Abfragedatum: 10.11.2011.

[Bal98] Helmut Balzert. *Lehrbuch der Software-Technik*. Spektrum, http://www.springer.com/spektrum+akademischer+verlag?SGWID=0-152602-0-0-0, 1998. ISBN 3-8274-0065-1. Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung.

[Ben04] Marcel Bennicke. *Messen im Software-Engineering und metrikbasierte Qualitätsanalyse*. Virtuells Software Engineering Kompetenzzentrum, 2004.

[Dum92] Reiner Dumke. *Softwareentwicklung nach Maß*. Vieweg, http://www.vieweg.de, 1992. ISBN 3-528-05232-5. Schätzen, Messen, Bewerten.

[Dum96] Reiner Dumke. *Software-Metriken in der Praxis*. Springer, http://www.springer.de, 1996. ISBN 3-540-60372-7. Einführung und Anwendung von Software-Metriken in der industriellen Praxis.

[Dum01] Reiner Dumke. *Software Engineering*. Vieweg, http://www.vieweg.de, 2001. ISBN 3-528-25355-X. Eine Einführung für Informatiker und Ingenieure: System, Erfahrungen, Methoden, Tools.

[Geb03] Klaus Gebhardt. *Qualitätsmanagement*. http://www.quality.de/lexikon/qualitaetsmanagement.htm, 2003. Abfragedatum: 11.10.2011.

[Jun02] Stefan Jungmayr. *Testability measurement and software dependencies*. testbarkeit.de, http://www.testbarkeit.de/Publikationen.html, 2002. Abfragedatum: 24.11.2011.

[KE10] Gernot Starke Karl Eilebrecht. *Patterns kompakt*. Spektrum, http://www.sepektrum.de, 2010. ISBN 978-3-8274-2525-6. Entwurfsmuster für effektive Software-Entwicklung.

[Kem93] Chidamber Kemerer. *A Metrics suite for Object Oriented design.* http://www.aivosto.com/project/help/pm-oo-ck.html, 1993. Abfragedatum: 28.02.2012.

[Mar02] Robert C. Martin. *Agile Software Development.* Prentice Hall, http://www.prenticehall.com, 2002. ISBN 978-0-13-597444-5. Principles, Patterns and Practices.

[Mar05] Robert C. Martin. *Working effectively with Legacy Code.* Prentice Hall, 2005. ISBN 0-13-117705-2.

[Mar09] Robert C. Martin. *Clean Code.* Pearson Education, http://www.pearson.co.uk, 2009. ISBN 978-0-13-235-088-4. A Handbook of Agile Software Craftsmanship.

[MBC07] Sandy Shrum Mary Beth Chrissis, Mike Konrad. *CMMI.* Addison-Wesley, http://www.addison-wesley.de, 2007. ISBN 978-3-446-8273-2784-0. Richtlinien für Prozess-Integration und Produkt-Verbesserung.

[Ojh94] Neeraj Ojha. *Object oriented metrics for early system characterization.* www.cs.clemson.edu/~johnmc/papers/METRICS/crcmetric.ps, 1994. Abfragedatum: 27.02.2012.

[Pre11] Prof. Dr. Christian Prehofer. *Methoden des Software Engineering.* Ludwig-Maximilians-Univerität München, http://www.pst.ifi.lmu.de/Lehre/wise-10-11/mse/materialien/folien-a-einfuehrung/view, 2011. ISBN 3-8273-7001-9. Pdf-Dokument - Abfragedatum: 18.08.2011.

[Rä04] Manfred Rätzmann. *Software-Testing und Internationalisierung.* Galileo Press, http://www.galileocomputing.de, 2004. ISBN 3-89842-539-8.

[RW12] Stefan Lieser Ralf Westphal. *Clean Code Developer.* http://clean-code-developer.de, 2012. Abfragedatum: 27.02.2012.

[Sch06] Kurt Schneider. *Abenteuer Softwarequalität.* dpunkt.verlag, http://www.dpunkt.de, 2006. ISBN 978-3-89864-472-3. Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement.

[Sne10] Harry M. Sneed. *Software in Zahlen.* Carl Hanser Verlag, http://www.hanser.de, 2010. ISBN 978-3-446-42175-2. Die Vermessung von Applikationen.

[Som01] Ian Sommerville. *Software Engineering.* Addison-Wesley, http://www.addison-wesley.de, 2001. ISBN 3-8273-7001-9. 6. Auflage.

[Tha93] Georg Erwin Thaller. *Qualitätsoptimierung der Softwareentwicklung.* Friedr. Vieweg Verlagsgesellschaft, 1993. ISBN 3-528-05287-2.

[Tha00a] Georg Erwin Thaller. *ISO 9001.* Verlag Heinz Heise GmbH und Co KG, http://www.heise-medien.de, 2000. ISBN 3-88229-181-8. Softwareentwicklung in der Praxis.

[Tha00b] Georg Erwin Thaller. *Software-Metriken.* Verlag Technik Berlin, 2000. ISBN 3-341-12060-5. Einsetzen, bewerten, messen.

[Wal00] Ernst Wallmüller. *Softwarequalitätsmanagement in der Praxis.* Hanser, http://www.hanser.de, 2000. ISBN 3-446-21367-8. Software-Qualität durch Führung und Verbesserung von Software-Prozessen.

[Zus98] Horst Zuse. *A Framework of Software Measurement.* Walter de Gruyter und Co, 1998. ISBN 3-11-012226-X.

Anhang A

Codebeispiele

A.1 Testbarkeit

A.1.1 Initialisierung von Klassenvariablen

Die Klasse ContextProcessor zeigt eine direkte Initialisierung der Klassenvariablen im Konstruktor. Diese Klassen können nicht gemockt und müssen demnach mitgetestet werden.

```
1  public class ContextProcessor implements EodProcessor {
2
3     private Log log = LogFactory.getLog(ContextProcessor.class);
4     private ContextDAO contextDAO;
5     private DeleteContext deleteContext;
6     private ReferentialIntegrity keepReferentialIntegrity;
7     private TimeProvider timeProvider;
8
9      public ContextProcessor(final StatementProcessor spIn) {
10       contextDAO = new ContextDAO(spIn);
11       deleteContext = new DeleteContext(spIn);
12       keepReferentialIntegrity = new ReferentialIntegrity(spIn);
13       timeProvider = new SystemTimeProvider();
14     }
15
16     @Override
17     public EodProtocol process(final EodConfiguration eodConfig) throws
           Exception {
18         EodProtocol eodProtocol = new EodProtocol();
19         ContextParamParser paramParser = new ContextParamParser();
20
21         String additionalParamData = eodConfig.getStoragePeriod(); //Other
               Format (A:90,B70,C90...)
22         ContextParamData parsedData = paramParser.parse(additionalParamData);
23         Collection<String> contextIds = getContextIds(parsedData);
```

```
24          if (isEmpty(contextIds)) {
25              log.info("No records found to delete...");
26              eodProtocol.setInfo("No records found to delete...");
27          } else {
28              log.info("Found " + contextIds.size() + " record(s) to delete..."
                    );
29              if (null == additionalParamData) {
30                  log.info("Deleting records older than " + parsedData.
                        getDefaultStoragePeriod() + " day(s) now...");
31                  eodProtocol.setInfo("Records older than " + parsedData.
                        getDefaultStoragePeriod() + " deleted");
32              } else {
33                  log.info("Deleting records based on parameter [" +
                        additionalParamData + "]");
34                  eodProtocol.setInfo("Records based on parameter [" +
                        additionalParamData + "] deleted");
35              }
36              deleteContext.process(contextIds);
37              deleteContextWithoutCustomer(parsedData.getDefaultStoragePeriod()
                    );
38              keepReferentialIntegrity.process();
39              eodProtocol.setInfo(contextIds.size() + " record(s) deleted");
40              eodProtocol.setDeleted(contextIds.size());
41          }
42          log.info("done...");
43          return eodProtocol;
44      }
45
46      @Override
47      public EodConfiguration getEodConfiguration() {
48          SimpleEodConfiguration simpleEodConfiguration = new
                SimpleEodConfiguration("90");
49          simpleEodConfiguration.setAdditionalParameter("90");
50          return simpleEodConfiguration;
51      }
52
53      private boolean isEmpty(final Collection<String> contextIds) {
54        return 0 == contextIds.size();
55      }
56
57    private Collection<String> getContextIds(final ContextParamData parsedData
            ) throws SQLException {
58        if (parsedData.sameForAllCustomers()) {
59           return contextDAO.getByTimeStampOpen(parsedData.
                getDefaultStoragePeriod());
60        } else {
61           return getContextIdsForEachGroup(parsedData);
62        }
63    }
64
65    private Collection<String> getContextIdsForEachGroup(final
            ContextParamData parsedData) throws SQLException {
66        Collection<String> contextIds;
67        Map<String, Integer> customerGroups = parsedData.getCustomerGroups();
68
69        contextIds = getContextIdsOnlyForSpecifiedGroups(customerGroups);
```

```
70        if (parsedData.deleteOtherGroups()) {
71           int storageperoid = parsedData.getDefaultStoragePeriod();
72           Set<String> groups = getGroupsToExclude(customerGroups);
73           Collection<String> ids = contextDAO.
                 getByTimestampOpenAndExludedGroups(storageperoid, groups,
                 timeProvider);
74           if (0 < ids.size()) {
75              contextIds.addAll(ids);
76           }
77        }
78        return contextIds;
79     }
80
81     private Set<String> getGroupsToExclude(final Map<String, Integer>
           customerGroups) {
82        Set<String> groups = new HashSet<String>();
83        for (Iterator it  = customerGroups.entrySet().iterator(); it.hasNext()
              ;) {
84           Map.Entry<String, Integer> entry = (Map.Entry) it.next();
85           groups.add(entry.getKey());
86        }
87        return groups;
88     }
89
90     private Collection<String> getContextIdsOnlyForSpecifiedGroups(final Map<
           String, Integer> customerGroups)
91              throws SQLException {
92        Collection<String> contextIds = new ArrayList<String>();
93        for (Iterator it  = customerGroups.entrySet().iterator(); it.hasNext()
              ;) {
94           Map.Entry<String, Integer> entry = (Map.Entry) it.next();
95           contextIds = contextDAO.getByGroupAndTimestampOpen(entry.getKey(),
                 entry.getValue(), timeProvider);
96        }
97        return contextIds;
98     }
99
100    private void deleteContextWithoutCustomer(final int defaultStoragePeriod)
           throws SQLException, EodQueryException {
101       Collection<String> ids = contextDAO.getByTimestampOpenAndEmptyGroup(
              defaultStoragePeriod, timeProvider);
102       deleteContext.process(ids);
103    }
104 }
```

Abbildung A.1: Die Klasse ContextProcessor

A.1.2 Initialisierungsblöcke

In Zeile 32 wird die Init-Methode aufgerufen. In Zeile 36 - 47 werden nun Klassenvariablen mittels Setter-Methoden initialisiert.

```
1  public class DatabaseOrderStorage implements OrderStorage {
2
3      private static final String NOT_IMPLEMENTED_YET = "Not implemented yet";
4      private static final long MICROS_PER_MINUTE = 60000000;
5
6      private Log log = LogFactory.getLog(DatabaseOrderStorage.class);
7      private LongTimestampProvider timestampProvider = new
           LongTimestampProvider(new SystemTimeProvider());
8      private Map<State, Iterator<InventoryItem>> ordersIterators = new HashMap
           <State, Iterator<InventoryItem>>();
9      private StopWatch stopWatch = new StopWatch();
10
11     private StatementProcessor statementProcessor;
12     private Connection connection;
13     private SubSysFactory subSysFactory;
14
15     private UploadBlockingPeriodDAO uploadBlockingPeriodDAO;
16     private ForeignBankConnectionDAO foreignBankConnectionDAO;
17     private ForeignDownloadDAO foreignDownloadDAO;
18     private OrderConverter orderConverter;
19     private Configuration config;
20     private InventoryProvider inventoryProvider;
21     private UpdateInventory updateInventory;
22
23
24     public DatabaseOrderStorage(
25       final SubSysFactory subSysFactoryParam,
26       final StatementProcessor statementProcessorParam,
27       final Connection connectionParam,
28       final ConverterFactory converterFactory) throws SQLException {
29         connection = connectionParam;
30         statementProcessor = statementProcessorParam;
31         subSysFactory = subSysFactoryParam;
32         init(converterFactory);
33         stopWatch.startNew();
34     }
35
36     void init(final ConverterFactory converterFactory) throws SQLException {
37         AuthorizationSubSys authorizationSubSys =
38         (AuthorizationSubSys) subSysFactory.getSubSys("authorization",
               connection.getClient());
39         setForeignBankConnectionDAO(new ForeignBankConnectionDAO(
               statementProcessor));
40         setUploadBlockingPeriodDAO(new UploadBlockingPeriodDAO(
               statementProcessor));
41         setForeignDownloadDAO(new ForeignDownloadDAO(statementProcessor));
42         setInventoryProvider(new InventoryProvider(authorizationSubSys,
               connection));
```

```
43          setUpdateInventory(new UpdateInventory(authorizationSubSys,
                connection));
44          setConfig(subSysFactory.getConfigurationProvider().getConfig(
                connection.getClient()));
45          setOrderConverter(new OrderConverter(statementProcessor, connection,
                authorizationSubSys,
46                  config, converterFactory));
47      }
```

Abbildung A.2: Initialisierungsblock von DatabaseOrderStorage

In Zeile 16 wird nun eine neue Klasse DatabaseOrderStorage erstellt, und in ihr die Methode *init* sowie *isSendable* überschrieben. Diese zwei Methoden können nun nicht mehr getestet werden. Von Zeile 26 - 28 werden Klassenvariablen, die für jeden Test benötigt werden, initialisiert.

```
1  public class DatabaseOrderStorageTest {
2
3      private final Bank bank = new Bank("Bank", "Bank");
4      private DatabaseOrderStorage sut;
5      private SubSysFactory subSysFactory;
6      private StatementProcessor statementProcessor;
7      private Configuration config;
8      private ConverterFactory converterFactory;
9      private Connection connection = new Connection("TestClient", new
           BankRelated("CustomerId"), new Id("ForeignCustomerId"),
10             new Id("ForeignUserId"), new Id("Channel"));
11     private InventoryProvider inventoryProvider = EasyMock.createMock(
           InventoryProvider.class);
12     private OrderConverter orderConverter = EasyMock.createMock(
           OrderConverter.class);
13
14     @Before
15     public void setUp() throws Exception {
16         sut = new DatabaseOrderStorage(subSysFactory, statementProcessor,
               connection, converterFactory) {
17             @Override
18             %\lstemb%void init%\lstnb%(final ConverterFactory
                   converterFactory) throws SQLException {
19             }
20
21             @Override
22             boolean isSendable(final InventoryItem item) throws Exception {
23                 return true;
24             }
25         };
26         %\lstemb%sut.setInventoryProvider(inventoryProvider);%\lstnb%
27         %\lstemb%sut.setConfig(config);%\lstnb%
28         %\lstemb%sut.setOrderConverter(orderConverter);%\lstnb%
29     }
```

Abbildung A.3: Testklasse von DatabaseOrderStorage

A.1.3 Doppelter Quellcode

In Abbildung A.4 ist die Klasse UserProcessor dargestellt. Dabei werden nur die Elemente gezeigt, die für ein Entfernen von Duplications berücksichtigt werden mussten.

```
1   public UserProcessor(final String clientParam, final StatementProcessor
        statementProcessorParam) {
2       client = clientParam;
3       statementProcessor = statementProcessorParam;
4       deleteIdProvider = new DeleteIdProvider(client, statementProcessor);
5   }
6
7   @Override
8   public EodProtocol process(final EodConfiguration eodConfig) throws
        EodException {
9       DeleteMethod deleteMethod = new DeleteMethod(eodConfig.
            getDeleteMethod());
10      checkDeleteMethod(deleteMethod);
11      EodProtocol eodProtocol = new EodProtocol();
12      statementProcessor.turnOffAutoCommit();
13      try {
14          String userStoragePeriod = eodConfig.getStoragePeriod();
15          Collection<User> users = getUsersToDelete(Integer.valueOf(
                userStoragePeriod));
16          if (isEmpty(users)) {
17              log.info("No Users found to delete...");
18              eodProtocol.setInfo("No Users found to delete...");
19          } else {
20              log.info("Found " + users.size() + " User(s) to delete");
21              log.info("Deleting records older than " + userStoragePeriod +
                     " day(s) now...");
22              processUsers(users, deleteMethod);
23              eodProtocol.setInfo(users.size() + " User(s) deleted");
24              eodProtocol.setDeleted(users.size());
25          }
26          statementProcessor.turnOnAutoCommit();
27          log.info("done ...");
28      } catch (SQLException e) {
29          handleException(e);
30      } catch (EodQueryException e) {
31          handleException(e);
32      }
33      return eodProtocol;
34  }
35
36  @Override
37  public EodConfiguration getEodConfiguration() {
38      return new SimpleEodConfiguration("1");
39  }
40
41  private void checkDeleteMethod(final DeleteMethod deleteMethod) throws
        EodException {
42      if (!deleteMethod.isKnown()) {
```

```
43              throw new EodException("Unknown delete method [" + deleteMethod +
                    "]");
44          }
45      }
46
47      private void handleException(final Throwable e) throws EodException {
48          log.error(e.getMessage(), e);
49          statementProcessor.rollback();
50          statementProcessor.turnOnAutoCommit();
51          throw new EodException(e.getMessage(), e);
52      }
```

Abbildung A.4: Klasse UserProcessor vor dem Refactoring

Das Ergebnis ist eine etwas schlankere Klasse. Auch hat die Methode *process* an Komplexität verloren.

```
1   public UserProcessor(final String clientParam, final StatementProcessor
        statementProcessorParam) {
2       client = clientParam;
3       statementProcessor = statementProcessorParam;
4       deleteIdProvider = new DeleteIdProvider(client, statementProcessor);
5   }
6
7   @Override
8   public EodProtocol process(final EodConfiguration eodConfig) throws
        Exception {
9       DeleteMethod deleteMethod = new DeleteMethod(eodConfig.
            getDeleteMethod());
10      EodProtocol eodProtocol = new EodProtocol();
11      String userStoragePeriod = eodConfig.getStoragePeriod();
12      Collection<User> users = getUsersToDelete(Integer.valueOf(
            userStoragePeriod));
13      if (isEmpty(users)) {
14          log.info("No Users found to delete...");
15          eodProtocol.setInfo("No Users found to delete...");
16      } else {
17          log.info("Found " + users.size() + " User(s) to delete");
18          log.info("Deleting records older than " + userStoragePeriod + "
                day(s) now...");
19          processUsers(users, deleteMethod);
20          eodProtocol.setInfo(users.size() + " User(s) deleted");
21          eodProtocol.setDeleted(users.size());
22      }
23      log.info("done ...");
24      return eodProtocol;
25  }
26
27  @Override
28  public EodConfiguration getEodConfiguration() {
29      return new SimpleEodConfiguration("1");
30  }
```

Abbildung A.5: Klasse UserProcessor nach dem Refactoring

Anhang B

Fallstudie

B.1 Die Metrik VOD

B.1.1 Law of Demeter und Methodenparameter

```
1  public class PasswordValidator implements EditFieldValidator {
2
3      private final Collection<PasswordAspect> passwordAspects = new ArrayList<
           PasswordAspect>();
4      private ClientEngine engine = null;
5
6      public PasswordValidator() {
7      }
8
9      public PasswordValidator(final ClientEngine clientEngine) {
10         engine = clientEngine;
11     }
12
13     /** {@inheritDoc} */
14     public boolean validate(final EditField editField) {
15         if (null == editField) {
16             return false;
17         }
18
19         setupPasswordAspects();
20         for (PasswordAspect passwordAspect : passwordAspects) {
21             if (!passwordAspect.validate(editField.getData())) {
22                 setValidationError(passwordAspect, editField);
23                 return false;
24             }
25         }
26
27         return true;
28     }
29
30     private void setupPasswordAspects() {
```

```
31          passwordAspects.clear();
32          if (engine != null) {
33              PasswordAspectConfig config = engine.getMandantConfiguration().
                    getPasswordAspectConfig();
34              if (config != null) {
35                  passwordAspects.addAll(config.getPasswordAspects());
36              }
37          }
38
39          if (passwordAspects.isEmpty()) {
40              setupDefaultPasswordAspects();
41          }
42      }
43
44      private void setupDefaultPasswordAspects() {
45          passwordAspects.add(new PasswordMinLengthAspect(5));
46      }
47
48      private void setValidationError(final PasswordAspect passwordAspect,
            final EditField editField) {
49          BaseForm topLevelForm = ((BaseForm) editField.getTopLevelForm());
50          if (passwordAspect instanceof PasswordMinLengthAspect) {
51              String[] params = {
52                      String.valueOf(((PasswordMinLengthAspect) passwordAspect)
                            .getParam())
53              };
54              topLevelForm.addValidationError(editField, passwordAspect.
                    getErrorMessage(), params);
55          } else if (passwordAspect instanceof PasswordMaxLengthAspect) {
56              String[] params = {
57                      String.valueOf(((PasswordMaxLengthAspect) passwordAspect)
                            .getParam())
58              };
59              topLevelForm.addValidationError(editField, passwordAspect.
                    getErrorMessage(), params);
60          } else {
61              topLevelForm.addValidationError(editField, passwordAspect.
                    getErrorMessage(), null);
62          }
63      }
64  }
```

Abbildung B.1: Die Klasse PasswordValidator unverändert

```
1
2       private final Collection<PasswordAspect> passwordAspects = new ArrayList<
            PasswordAspect>();
3       private PasswordAspectConfig config;
4
5       public PasswordValidator() {
6       }
7
8       public PasswordValidator(final PasswordAspectConfig configIn) {
9           config = configIn;
10      }
11
12      /** {@inheritDoc} */
13      public boolean validate(final EditField editField) {
14          if (null == editField) {
15              return false;
16          }
17
18          setupPasswordAspects();
19          for (PasswordAspect passwordAspect : passwordAspects) {
20              if (!passwordAspect.validate(editField.getData())) {
21                  setValidationError(passwordAspect, editField);
22                  return false;
23              }
24          }
25
26          return true;
27      }
28
29      private void setupPasswordAspects() {
30          passwordAspects.clear();
31          passwordAspects.addAll(config.getPasswordAspects());
32          if (passwordAspects.isEmpty()) {
33              setupDefaultPasswordAspects();
34          }
35      }
36
37      private void setupDefaultPasswordAspects() {
38          passwordAspects.add(new PasswordMinLengthAspect(5));
39      }
40
41      private void setValidationError(final PasswordAspect passwordAspect,
            final EditField editField) {
42          BaseForm topLevelForm = ((BaseForm) editField.getTopLevelForm());
43          if (passwordAspect instanceof PasswordMinLengthAspect) {
44              String[] params = {
45                      String.valueOf(((PasswordMinLengthAspect) passwordAspect)
                            .getParam())
46              };
47              topLevelForm.addValidationError(editField, passwordAspect.
                    getErrorMessage(), params);
48          } else if (passwordAspect instanceof PasswordMaxLengthAspect) {
49              String[] params = {
50                      String.valueOf(((PasswordMaxLengthAspect) passwordAspect)
                            .getParam())
51              };
```

```
52              topLevelForm.addValidationError(editField, passwordAspect.
                    getErrorMessage(), params);
53          } else {
54              topLevelForm.addValidationError(editField, passwordAspect.
                    getErrorMessage(), null);
55          }
56      }
```

Abbildung B.2: Die Klasse PasswordValidator refaktorisiert

B.2 Die Metrik LCOM4

```
1  public class ChangeKey2 extends ChangeKey {
2
3      public ChangeKey2() {
4          super(2);
5      }
6
7      /**
8       * {@inheritDoc}
9       */
10     @Override
11     protected void readBody(final Elements elements) {
12         setMessageReferenceCoded(elements.read(new Num(1)));
13         setCodeForFunctionType(elements.read(new Num(3)));
14         setSecurityProfile(SecurityProfile.RDH1);
15         setKeyName(elements.read(new KeyName()));
16         setPublicKey(elements.read(new PublicKey()));
17         setCertificate(elements.read(new Certificate()));
18     }
19
20     /**
21      * {@inheritDoc}
22      */
23     @Override
24     protected void writeBody(final List<Element> elements) {
25         elements.add(getMessageReferenceCoded());
26         elements.add(getCodeForFunctionType());
27         elements.add(getKeyName());
28         elements.add(getPublicKey());
29         elements.add(getCertificate());
30     }
31
32     /**
33      * {@inheritDoc}
34      */
35     @Override
36     public void assertValidSecurityProfile(final SecurityProfile
           securityProfile) {
37         if (!SecurityProfile.RDH1.equals(securityProfile)) {
38             throw new IllegalArgumentException("Invalid security profile for
                   HKSAK::2. [" + securityProfile + "].");
39         }
40     }
41
42 }
```

Abbildung B.3: Die Klasse ChangeKey2

```
1  public class ChangeKey3 extends ChangeKey {
2
3      public ChangeKey3() {
4          super(3);
5      }
6
7      /**
8       * {@inheritDoc}
9       */
10     @Override
11     protected void readBody(final Elements elements) {
12         setMessageReferenceCoded(elements.read(new Num(1)));
13         setCodeForFunctionType(elements.read(new Num(3)));
14         setSecurityProfile(elements.read(new SecurityProfile()));
15         setKeyName(elements.read(new KeyName()));
16         setPublicKey(elements.read(new PublicKey()));
17         setCertificate(elements.read(new Certificate()));
18     }
19
20     /**
21      * {@inheritDoc}
22      */
23     @Override
24     protected void writeBody(final List<Element> elements) {
25         elements.add(getMessageReferenceCoded());
26         elements.add(getCodeForFunctionType());
27         elements.add(getSecurityProfile());
28         elements.add(getKeyName());
29         elements.add(getPublicKey());
30         elements.add(getCertificate());
31     }
32
33     /**
34      * {@inheritDoc}
35      */
36     @Override
37     public void assertValidSecurityProfile(final SecurityProfile
           securityProfile) {
38         // nothing to do
39     }
40
41 }
```

Abbildung B.4: Die Klasse ChangeKey3

Anhang C

Coding-Rules

Es folgen alle festgelegten Coding-Rules, die Verwendung finden. Hierbei werden Blocker, Critical, Major, Minor und Info-Violations unterschieden. Diese sollen bei der Berechnung der Testbarkeit nicht berücksichtigt werden, finden jedoch Verwendung im Build-Prozess.

title	plugin
Array Type Style	checkstyle
Avoid Array Loops	pmd
Avoid Assert As Identifier	pmd
Avoid Calling Finalize	pmd
Avoid Catching NPE	pmd
Avoid Catching Throwable	pmd
Avoid Decimal Literals In Big Decimal Constructor	pmd
Avoid Duplicate Literals	pmd
Avoid Enum As Identifier	pmd
Avoid Inline Conditionals	checkstyle
Avoid Instanceof Checks In Catch Clause	pmd
Avoid Nested Blocks	checkstyle
Avoid Print Stack Trace	pmd
Avoid Rethrowing Exception	pmd
Avoid Star Import	checkstyle
Avoid Throwing Null Pointer Exception	pmd
Bad practice - Abstract class defines covariant compareTo() method	findbugs
Bad practice - Abstract class defines covariant equals() method	findbugs
Bad practice - Certain swing methods needs to be invoked in Swing thread	findbugs
Bad practice - Check for sign of bitwise operation	findbugs
Bad practice - Class defines clone() but doesn't implement Cloneable	findbugs
Bad practice - Class defines compareTo(...) and uses Object.equals()	findbugs
Bad practice - Class defines equals() and uses Object.hashCode()	findbugs
Bad practice - Class defines equals() but not hashCode()	findbugs
Bad practice - Class defines hashCode() and uses Object.equals()	findbugs
Bad practice - Class defines hashCode() but not equals()	findbugs
Bad practice - Class implements Cloneable but does not define or use clone method	findbugs
Bad practice - Class inherits equals() and uses Object.hashCode()	findbugs
Bad practice - Class is Externalizable but doesn't define a void constructor	findbugs
Bad practice - Class is Serializable but its superclass doesn't define a void constructor	findbugs
Bad practice - Class is Serializable, but doesn't define serialVersionUID	findbugs
Bad practice - Class is not derived from an Exception, even though it is named as such	findbugs
Bad practice - Class names shouldn't shadow simple name of implemented interface	findbugs
Bad practice - Class names shouldn't shadow simple name of superclass	findbugs
Bad practice - Classloaders should only be created inside doPrivileged block	findbugs
Bad practice - Clone method may return null	findbugs
Bad practice - Comparator doesn't implement Serializable	findbugs
Bad practice - Comparison of String objects using == or !=	findbugs
Bad practice - Comparison of String parameter using == or !=	findbugs
Bad practice - Confusing method names	findbugs
Bad practice - Covariant compareTo() method defined	findbugs
Bad practice - Covariant equals() method defined	findbugs
Bad practice - Creates an empty jar file entry	findbugs
Bad practice - Creates an empty zip file entry	findbugs
Bad practice - Dubious catching of IllegalMonitorStateException	findbugs
Bad practice - Empty finalizer should be deleted	findbugs
Bad practice - Equals checks for noncompatible operand	findbugs
Bad practice - Equals method should not assume anything about the type of its argument	findbugs
Bad practice - Explicit invocation of finalizer	findbugs
Bad practice - Fields of immutable classes should be final	findbugs
Bad practice - Finalizer does not call superclass finalizer	findbugs
Bad practice - Finalizer does nothing but call superclass finalizer	findbugs
Bad practice - Finalizer nullifies superclass finalizer	findbugs
Bad practice - Finalizer nulls fields	findbugs
Bad practice - Finalizer only nulls fields	findbugs

Bad practice - Iterator next() method can't throw NoSuchElementException	findbugs
Bad practice - Method doesn't override method in superclass due to wrong package for param	findbugs
Bad practice - Method ignores exceptional return value	findbugs
Bad practice - Method ignores results of InputStream.read()	findbugs
Bad practice - Method ignores results of InputStream.skip()	findbugs
Bad practice - Method invoked that should be only be invoked inside a doPrivileged block	findbugs
Bad practice - Method invokes System.exit(...)	findbugs
Bad practice - Method invokes dangerous method runFinalizersOnExit	findbugs
Bad practice - Method may fail to close database resource	findbugs
Bad practice - Method may fail to close database resource on exception	findbugs
Bad practice - Method may fail to close stream	findbugs
Bad practice - Method may fail to close stream on exception	findbugs
Bad practice - Method might drop exception	findbugs
Bad practice - Method might ignore exception	findbugs
Bad practice - Method with Boolean return type returns explicit null	findbugs
Bad practice - Needless instantiation of class that only supplies static methods	findbugs
Bad practice - Non-serializable class has a serializable inner class	findbugs
Bad practice - Non-serializable value stored into instance field of a serializable class	findbugs
Bad practice - Random object created and used only once	findbugs
Bad practice - Serializable inner class	findbugs
Bad practice - Static initializer creates instance before all static final fields assigned	findbugs
Bad practice - Store of non serializable object into HttpSession	findbugs
Bad practice - Superclass uses subclass during initialization	findbugs
Bad practice - Suspicious reference comparison	findbugs
Bad practice - The readResolve method must be declared with a return type of Object.	findbugs
Bad practice - Transient field that isn't set by deserialization.	findbugs
Bad practice - Unchecked type in generic call	findbugs
Bad practice - Usage of GetResource may be unsafe if class is extended	findbugs
Bad practice - Use of identifier that is a keyword in later versions of Java	findbugs
Bad practice - Use of identifier that is a keyword in later versions of Java	findbugs
Bad practice - Very confusing method names (but perhaps intentional)	findbugs
Bad practice - clone method does not call super.clone()	findbugs
Bad practice - equals method fails for subtypes	findbugs
Bad practice - equals() method does not check for null argument	findbugs
Bad practice - serialVersionUID isn't final	findbugs
Bad practice - serialVersionUID isn't long	findbugs
Bad practice - serialVersionUID isn't static	findbugs
Bad practice - toString method may return null	findbugs
Big Integer Instantiation	pmd
Boolean Instantiation	pmd
Broken Null Check	pmd
Class Cast Exception With To Array	pmd
Clone Throws Clone Not Supported Exception	pmd
Collapsible If Statements	pmd
Comment pattern matcher	checkstyle
Compare Objects With Equals	pmd
Constant Name	checkstyle
Constructor Calls Overridable Method	pmd
Correctness - "." used for regular expression	findbugs
Correctness - A collection is added to itself	findbugs
Correctness - A known null value is checked to see if it is an instance of a type	findbugs
Correctness - A parameter is dead upon entry to a method but overwritten	findbugs
Correctness - An apparent infinite loop	findbugs
Correctness - An apparent infinite recursive loop	findbugs
Correctness - Apparent method/constructor confusion	findbugs
Correctness - Array formatted in useless way using format string	findbugs

Correctness - Bad attempt to compute absolute value of signed 32-bit hashcode	findbugs
Correctness - Bad attempt to compute absolute value of signed 32-bit random integer	findbugs
Correctness - Bad comparison of nonnegative value with negative constant	findbugs
Correctness - Bad comparison of signed byte	findbugs
Correctness - Bad constant value for month	findbugs
Correctness - Bitwise OR of signed byte value	findbugs
Correctness - Bitwise add of signed byte value	findbugs
Correctness - Call to equals() comparing different interface types	findbugs
Correctness - Call to equals() comparing different types	findbugs
Correctness - Call to equals() comparing unrelated class and interface	findbugs
Correctness - Call to equals() with null argument	findbugs
Correctness - Can't use reflection to check for presence of annotation without runtime retentio	findbugs
Correctness - Check for sign of bitwise operation	findbugs
Correctness - Check to see if ((...) & 0) == 0	findbugs
Correctness - Class defines field that masks a superclass field	findbugs
Correctness - Class defines static field that appears to allow memory bloat [fb-contrib]	findbugs
Correctness - Class overrides a method implemented in super class Adapter wrongly	findbugs
Correctness - Class relies on internal api classes [fb-contrib]	findbugs
Correctness - Collections should not contain themselves	findbugs
Correctness - Covariant equals() method defined for enum	findbugs
Correctness - Covariant equals() method defined, Object.equals(Object) inherited	findbugs
Correctness - Creation of ScheduledThreadPoolExecutor with zero core threads	findbugs
Correctness - Dead store of class literal	findbugs
Correctness - Don't use removeAll to clear a collection	findbugs
Correctness - Doomed attempt to append to an object output stream	findbugs
Correctness - Doomed test for equality to NaN	findbugs
Correctness - Double assignment of field	findbugs
Correctness - Double.longBitsToDouble invoked on an int	findbugs
Correctness - Exception created and dropped rather than thrown	findbugs
Correctness - Explicit annotation inconsistent with use	findbugs
Correctness - Explicit annotation inconsistent with use	findbugs
Correctness - Field only ever set to null	findbugs
Correctness - File.separator used for regular expression	findbugs
Correctness - Format string placeholder incompatible with passed argument	findbugs
Correctness - Format string references missing argument	findbugs
Correctness - Futile attempt to change max pool size of ScheduledThreadPoolExecutor	findbugs
Correctness - Illegal format string	findbugs
Correctness - Impossible cast	findbugs
Correctness - Incompatible bit masks	findbugs
Correctness - Incompatible bit masks	findbugs
Correctness - Integer multiply of result of integer remainder	findbugs
Correctness - Integer remainder modulo 1	findbugs
Correctness - Integer shift by an amount not in the range 0..31	findbugs
Correctness - Invalid syntax for regular expression	findbugs
Correctness - Invocation of equals() on an array, which is equivalent to ==	findbugs
Correctness - Invocation of hashCode on an array	findbugs
Correctness - Invocation of toString on an array	findbugs
Correctness - Invocation of toString on an array	findbugs
Correctness - JUnit assertion in run method will not be noticed by JUnit	findbugs
Correctness - Method assigns boolean literal in boolean expression	findbugs
Correctness - Method attempts to access a prepared statement parameter with index 0	findbugs
Correctness - Method attempts to access a result set field with index 0	findbugs
Correctness - Method call passes null for nonnull parameter	findbugs
Correctness - Method call passes null to a nonnull parameter	findbugs
Correctness - Method calls BigDecimal.equals() [fb-contrib]	findbugs
Correctness - Method defines a variable that obscures a field	findbugs

Correctness - Method does not check for null argument	findbugs
Correctness - Method doesn't override method in superclass due to wrong package for param	findbugs
Correctness - Method encodes String bytes without specifying the character encoding [fb-conti	findbugs
Correctness - Method ignores return value	findbugs
Correctness - Method ignores return value	findbugs
Correctness - Method may return null, but is declared @NonNull	findbugs
Correctness - Method must be private in order for serialization to work	findbugs
Correctness - Method passes a negative number as a bit to a BitSet which isn't supported [fb-c	findbugs
Correctness - Method passes a non array object to a parameter that expects an array [fb-contr	findbugs
Correctness - Method performs math using floating point precision	findbugs
Correctness - Method returns an array that appears not to be initialized [fb-contrib]	findbugs
Correctness - More arguments are passed that are actually used in the format string	findbugs
Correctness - No previous argument for format string	findbugs
Correctness - No relationship between generic parameter and method argument	findbugs
Correctness - Non derivable method declares throwing an exception that isn't thrown [fb-contri	findbugs
Correctness - Non-virtual method call passes null for nonnull parameter	findbugs
Correctness - Nonsensical self computation involving a field (e.g., x & x)	findbugs
Correctness - Nonsensical self computation involving a variable (e.g., x & x)	findbugs
Correctness - Null pointer dereference	findbugs
Correctness - Null pointer dereference in method on exception path	findbugs
Correctness - Null value is guaranteed to be dereferenced	findbugs
Correctness - Nullcheck of value previously dereferenced	findbugs
Correctness - Number of format-string arguments does not correspond to number of placehold	findbugs
Correctness - Overwritten increment	findbugs
Correctness - Possible null pointer dereference	findbugs
Correctness - Possible null pointer dereference in method on exception path	findbugs
Correctness - Primitive array passed to function expecting a variable number of object argume	findbugs
Correctness - Primitive value is unboxed and coerced for ternary operator	findbugs
Correctness - Random value from 0 to 1 is coerced to the integer 0	findbugs
Correctness - Repeated conditional tests	findbugs
Correctness - Self assignment of field	findbugs
Correctness - Self comparison of field with itself	findbugs
Correctness - Self comparison of value with itself	findbugs
Correctness - Signature declares use of unhashable class in hashed construct	findbugs
Correctness - Static Thread.interrupted() method invoked on thread instance	findbugs
Correctness - Store of null value into field annotated NonNull	findbugs
Correctness - TestCase declares a bad suite method	findbugs
Correctness - TestCase defines setUp that doesn't call super.setUp()	findbugs
Correctness - TestCase defines tearDown that doesn't call super.tearDown()	findbugs
Correctness - TestCase has no tests	findbugs
Correctness - TestCase implements a non-static suite method	findbugs
Correctness - The readResolve method must not be declared as a static method.	findbugs
Correctness - The type of a supplied argument doesn't match format specifier	findbugs
Correctness - Uncallable method defined in anonymous class	findbugs
Correctness - Uninitialized read of field in constructor	findbugs
Correctness - Unnecessary type check done using instanceof operator	findbugs
Correctness - Unneeded use of currentThread() call, to call interrupted()	findbugs
Correctness - Use of class without a hashCode() method in a hashed data structure	findbugs
Correctness - Useless assignment in return statement	findbugs
Correctness - Useless control flow to next line	findbugs
Correctness - Using pointer equality to compare different types	findbugs
Correctness - Vacuous call to collections	findbugs
Correctness - Value annotated as carrying a type qualifier used where a value that must not ca	findbugs
Correctness - Value annotated as never carrying a type qualifier used where value carrying tha	findbugs
Correctness - Value is null and guaranteed to be dereferenced on exception path	findbugs
Correctness - Value that might carry a type qualifier is always used in a way prohibits it from ha	findbugs

Correctness - Value that might not carry a type qualifier is always used in a way requires that t	findbugs
Correctness - Very confusing method names	findbugs
Correctness - equals method always returns false	findbugs
Correctness - equals method always returns true	findbugs
Correctness - equals method compares class names rather than class objects	findbugs
Correctness - equals method overrides equals in superclass and may not be symmetric	findbugs
Correctness - equals() method defined that doesn't override Object.equals(Object)	findbugs
Correctness - equals() method defined that doesn't override equals(Object)	findbugs
Correctness - equals() used to compare array and nonarray	findbugs
Correctness - hasNext method invokes next	findbugs
Correctness - instanceof will always return false	findbugs
Correctness - int value cast to double and then passed to Math.ceil	findbugs
Correctness - int value cast to float and then passed to Math.round	findbugs
Cyclomatic Complexity	checkstyle
Dodgy - Ambiguous invocation of either an inherited or outer method	findbugs
Dodgy - Call to unsupported method	findbugs
Dodgy - Check for oddness that won't work for negative numbers	findbugs
Dodgy - Class exposes synchronization and semaphores in its public interface	findbugs
Dodgy - Class extends Servlet class and uses instance variables	findbugs
Dodgy - Class extends Struts Action class and uses instance variables	findbugs
Dodgy - Class implements same interface as superclass	findbugs
Dodgy - Class is final but declares protected field	findbugs
Dodgy - Class too big for analysis	findbugs
Dodgy - Code contains a hard coded reference to an absolute pathname	findbugs
Dodgy - Complicated, subtle or wrong increment in for-loop	findbugs
Dodgy - Computation of average could overflow	findbugs
Dodgy - Consider returning a zero length array rather than null	findbugs
Dodgy - Dead store of null to local variable	findbugs
Dodgy - Dead store to local variable	findbugs
Dodgy - Dereference of the result of readLine() without nullcheck	findbugs
Dodgy - Double assignment of local variable	findbugs
Dodgy - Exception is caught when Exception is not thrown	findbugs
Dodgy - Immediate dereference of the result of readLine()	findbugs
Dodgy - Initialization circularity	findbugs
Dodgy - Invocation of substring(0), which returns the original value	findbugs
Dodgy - Load of known null value	findbugs
Dodgy - Method checks to see if result of String.indexOf is positive	findbugs
Dodgy - Method directly allocates a specific implementation of xml interfaces	findbugs
Dodgy - Method discards result of readLine after checking if it is nonnull	findbugs
Dodgy - Method uses the same code for two branches	findbugs
Dodgy - Method uses the same code for two switch clauses	findbugs
Dodgy - Non serializable object written to ObjectOutput	findbugs
Dodgy - Non-Boolean argument formatted using %b format specifier	findbugs
Dodgy - Parameter must be nonnull but is marked as nullable	findbugs
Dodgy - Possible null pointer dereference due to return value of called method	findbugs
Dodgy - Possible null pointer dereference on path that might be infeasible	findbugs
Dodgy - Potentially dangerous use of non-short-circuit logic	findbugs
Dodgy - Questionable cast to abstract collection	findbugs
Dodgy - Questionable cast to concrete collection	findbugs
Dodgy - Questionable use of non-short-circuit logic	findbugs
Dodgy - Redundant comparison of non-null value to null	findbugs
Dodgy - Redundant comparison of two null values	findbugs
Dodgy - Redundant nullcheck of value known to be non-null	findbugs
Dodgy - Redundant nullcheck of value known to be null	findbugs
Dodgy - Remainder of 32-bit signed random integer	findbugs
Dodgy - Remainder of hashCode could be negative	findbugs

Dodgy - Result of integer multiplication cast to long	findbugs
Dodgy - Self assignment of local variable	findbugs
Dodgy - Test for floating point equality	findbugs
Dodgy - Thread passed where Runnable expected	findbugs
Dodgy - Transient field of class that isn't Serializable.	findbugs
Dodgy - Unchecked/unconfirmed cast	findbugs
Dodgy - Unsigned right shift cast to short/byte	findbugs
Dodgy - Unusual equals method	findbugs
Dodgy - Useless control flow	findbugs
Dodgy - Vacuous bit mask operation on integer value	findbugs
Dodgy - Vacuous comparison of integer value	findbugs
Dodgy - Write to static field from instance method	findbugs
Dodgy - instanceof will always return true	findbugs
Dodgy - int division result cast to double or float	findbugs
Dodgy - private readResolve method not inherited by subclasses	findbugs
Dont Import Java Lang	pmd
Dont Import Sun	pmd
Double Checked Locking	checkstyle
Empty Block	checkstyle
Empty Catch Block	pmd
Empty Finalizer	pmd
Empty Finally Block	pmd
Empty For Iterator Pad	checkstyle
Empty If Stmt	pmd
Empty Statement	checkstyle
Empty Static Initializer	pmd
Empty Switch Statements	pmd
Empty Synchronized Block	pmd
Empty Try Block	pmd
Empty While Stmt	pmd
Equals Hash Code	checkstyle
Equals Null	pmd
Exception As Flow Control	pmd
File Length	checkstyle
File Tab Character	checkstyle
Final Class	checkstyle
Final Field Could Be Static	pmd
Final Parameters	checkstyle
Finalize Does Not Call Super Finalize	pmd
Finalize Overloaded	pmd
For Loops Must Use Braces	pmd
Hidden Field	checkstyle
Hide Utility Class Constructor	checkstyle
Idempotent Operations	pmd
If Else Stmts Must Use Braces	pmd
If Stmts Must Use Braces	pmd
Illegal Import	checkstyle
Illegal Instantiation	checkstyle
Inefficient String Buffering	pmd
Inner Assignment	checkstyle
Instantiation To Get Class	pmd
Integer Instantiation	pmd
Interface Is Type	checkstyle
Internationalization - Consider using Locale parameterized version of invoked method	findbugs
JavaNCSS	checkstyle
Left Curly	checkstyle

Line Length	checkstyle
Local Final Variable Name	checkstyle
Local Variable Name	checkstyle
Loose coupling	pmd
Magic Number	checkstyle
Malicious code vulnerability - Field is a mutable Hashtable	findbugs
Malicious code vulnerability - Field is a mutable array	findbugs
Malicious code vulnerability - Field isn't final and can't be protected from malicious code	findbugs
Malicious code vulnerability - Field isn't final but should be	findbugs
Malicious code vulnerability - Field should be both final and package protected	findbugs
Malicious code vulnerability - Field should be moved out of an interface and made package pr	findbugs
Malicious code vulnerability - Field should be package protected	findbugs
Malicious code vulnerability - Finalizer should be protected, not public	findbugs
Malicious code vulnerability - May expose internal representation by incorporating reference to	findbugs
Malicious code vulnerability - May expose internal representation by returning reference to mu	findbugs
Malicious code vulnerability - May expose internal static state by storing a mutable object into	findbugs
Malicious code vulnerability - Public static method may expose internal representation by retur	findbugs
Member name	checkstyle
Method Length	checkstyle
Method Name	checkstyle
Method Param Pad	checkstyle
Missing Static Method In Non Instantiatable Class	pmd
Missing Switch Default	checkstyle
Modifier Order	checkstyle
More Than One Logger	pmd
Multithreaded correctness - A thread was created using the default empty run method	findbugs
Multithreaded correctness - A volatile reference to an array doesn't treat the array elements as	findbugs
Multithreaded correctness - Call to static Calendar	findbugs
Multithreaded correctness - Call to static DateFormat	findbugs
Multithreaded correctness - Class's readObject() method is synchronized	findbugs
Multithreaded correctness - Class's writeObject() method is synchronized but nothing else is	findbugs
Multithreaded correctness - Condition.await() not in loop	findbugs
Multithreaded correctness - Constructor invokes Thread.start()	findbugs
Multithreaded correctness - Empty synchronized block	findbugs
Multithreaded correctness - Field not guarded against concurrent access	findbugs
Multithreaded correctness - Inconsistent synchronization	findbugs
Multithreaded correctness - Inconsistent synchronization	findbugs
Multithreaded correctness - Incorrect lazy initialization and update of static field	findbugs
Multithreaded correctness - Incorrect lazy initialization of static field	findbugs
Multithreaded correctness - Invokes run on a thread (did you mean to start it instead?)	findbugs
Multithreaded correctness - Method calls Thread.sleep() with a lock held	findbugs
Multithreaded correctness - Method does not release lock on all exception paths	findbugs
Multithreaded correctness - Method does not release lock on all paths	findbugs
Multithreaded correctness - Method spins on field	findbugs
Multithreaded correctness - Method synchronizes on an updated field	findbugs
Multithreaded correctness - Mismatched notify()	findbugs
Multithreaded correctness - Mismatched wait()	findbugs
Multithreaded correctness - Monitor wait() called on Condition	findbugs
Multithreaded correctness - Mutable servlet field	findbugs
Multithreaded correctness - Naked notify	findbugs
Multithreaded correctness - Possible double check of field	findbugs
Multithreaded correctness - Static Calendar	findbugs
Multithreaded correctness - Static DateFormat	findbugs
Multithreaded correctness - Sychronization on getClass rather than class literal	findbugs
Multithreaded correctness - Synchronization on Boolean could lead to deadlock	findbugs
Multithreaded correctness - Synchronization on boxed primitive could lead to deadlock	findbugs

Multithreaded correctness - Synchronization on boxed primitive values	findbugs
Multithreaded correctness - Synchronization on field in futile attempt to guard that field	findbugs
Multithreaded correctness - Synchronization on interned String could lead to deadlock	findbugs
Multithreaded correctness - Synchronization performed on java.util.concurrent Lock	findbugs
Multithreaded correctness - Synchronize and null check on the same field.	findbugs
Multithreaded correctness - Unconditional wait	findbugs
Multithreaded correctness - Unsynchronized get method, synchronized set method	findbugs
Multithreaded correctness - Using notify() rather than notifyAll()	findbugs
Multithreaded correctness - Wait not in loop	findbugs
Multithreaded correctness - Wait with two locks held	findbugs
Naming - Avoid dollar signs	pmd
Naming - Class naming conventions	pmd
Naming - Method with same name as enclosing class	pmd
Naming - Suspicious Hashcode method name	pmd
Naming - Suspicious constant field name	pmd
Naming - Suspicious equals method name	pmd
Ncss Method Count	pmd
Ncss Type Count	pmd
Need Braces	checkstyle
Newline At End Of File	checkstyle
No Whitespace After	checkstyle
No Whitespace Before	checkstyle
Operator Wrap	checkstyle
Package name	checkstyle
Parameter Name	checkstyle
Parameter Number	checkstyle
Paren Pad	checkstyle
Performance - Could be refactored into a named static inner class	findbugs
Performance - Could be refactored into a static inner class	findbugs
Performance - Explicit garbage collection; extremely dubious except in benchmarking code	findbugs
Performance - Huge string constants is duplicated across multiple class files	findbugs
Performance - Inefficient use of keySet iterator instead of entrySet iterator	findbugs
Performance - Maps and sets of URLs can be performance hogs	findbugs
Performance - Method allocates a boxed primitive just to call toString	findbugs
Performance - Method allocates an object, only to get the class object	findbugs
Performance - Method calls static Math class method on a constant value	findbugs
Performance - Method concatenates strings using + in a loop	findbugs
Performance - Method invokes inefficient Boolean constructor; use Boolean.valueOf(...) instea	findbugs
Performance - Method invokes inefficient Number constructor; use static valueOf instead	findbugs
Performance - Method invokes inefficient floating-point Number constructor; use static valueOf	findbugs
Performance - Method invokes inefficient new String() constructor	findbugs
Performance - Method invokes inefficient new String(String) constructor	findbugs
Performance - Method invokes toString() method on a String	findbugs
Performance - Method passes simple concatenating string in StringBuffer or StringBuilder app	findbugs
Performance - Method uses toArray() with zero-length array argument	findbugs
Performance - Primitive value is boxed and then immediately unboxed	findbugs
Performance - Primitive value is boxed then unboxed to perform primitive coercion	findbugs
Performance - Private method is never called	findbugs
Performance - Should be a static inner class	findbugs
Performance - The equals and hashCode methods of URL are blocking	findbugs
Performance - Unread field	findbugs
Performance - Unread field: should this field be static?	findbugs
Performance - Unused field	findbugs
Performance - Use the nextInt method of Random rather than nextDouble to generate a rando	findbugs
Preserve Stack Trace	pmd
Redundant Modifier	checkstyle

Redundant Throws	checkstyle
Redundant import	checkstyle
Regexp Header	checkstyle
Regexp Singleline Java	checkstyle
Replace Enumeration With Iterator	pmd
Replace Hashtable With Map	pmd
Replace Vector With List	pmd
Right Curly	checkstyle
Security - A prepared statement is generated from a nonconstant String	findbugs
Security - Array is stored directly	pmd
Security - Empty database password	findbugs
Security - HTTP Response splitting vulnerability	findbugs
Security - HTTP cookie formed from untrusted input	findbugs
Security - Hardcoded constant database password	findbugs
Security - JSP reflected cross site scripting vulnerability	findbugs
Security - Nonconstant string passed to execute method on an SQL statement	findbugs
Security - Servlet reflected cross site scripting vulnerability	findbugs
Security - Servlet reflected cross site scripting vulnerability	findbugs
Simplify Boolean Expression	checkstyle
Simplify Boolean Return	checkstyle
Simplify Conditional	pmd
Simplify boolean returns	pmd
Singular Field	pmd
Static Variable Name	checkstyle
Strict Exception - Avoid throwing new instance of same exception	pmd
String Buffer Instantiation With Char	pmd
String Instantiation	pmd
String To String	pmd
Type Name	checkstyle
Typecast Paren Pad	checkstyle
Unconditional If Statement	pmd
Unnecessary Case Change	pmd
Unnecessary Local Before Return	pmd
Unnecessary Return	pmd
Unused Imports	checkstyle
Unused Null Check In Equals	pmd
Unused Private Field	pmd
Unused formal parameter	pmd
Unused imports	pmd
Unused local variable	pmd
Upper Ell	checkstyle
Use Array List Instead Of Vector	pmd
Use Arrays As List	pmd
Use Correct Exception Logging	pmd
Use Index Of Char	pmd
Use String Buffer Length	pmd
Useless Operation On Immutable	pmd
Useless Overriding Method	pmd
Useless String Value Of	pmd
Visibility Modifier	checkstyle
While Loops Must Use Braces	pmd
Whitespace After	checkstyle
Whitespace Around	checkstyle

Anhang D

Installation des Plugins

Es soll kurz vorgestellt werden, wie das Sonar-Plugin installiert werden kann. Vorausgesetzt wird, dass Sonar bereits als lauffähiges System vorhanden ist. Eine entsprechende Dokumentation befindet sich auf http://www.sonarsource.org.

D.1 Installation der Jar-Files

D.1.1 Sonar-Plugin

Das Sonar-Plugin befindet sich auf der CD-ROM als Jar-File. Diese Datei muss in den Ordner

```
$:sonar-home/extensions/plugins/
```

kopiert werden.

D.1.2 Sonar-Pitest-Plugin

Das Sonar-Pitest-Plugin befindet sich ebenfalls auf der CD-ROM. Diese Datei muss in den Ordner

```
$:sonar-home/extensions/plugins/
```

kopiert werden.

D.2 Konfiguration des Sonar-Plugins

Für den Einsatz des Sonar-Plugins ist zuvor eine Konfiguration notwendig. Hierfür sind Administrationsrechte erforderlich. Standardmäßig erhält man mit

```
Login: admin
Passwort: admin
```

Administrationsrechte.

Unter

```
Configuration -> Profile
```

müssen nun die Violations

- Bad Dependency
- Complexity Limit Overrun
- Law of Demeter
- LCOM4 Limit Overrun
- Survived mutant
- Switch Density

aktiviert werden.

Unter

```
Configuration -> General Settings -> XSP
```

sind alle individuellen Parameter des Plugins zu finden.

Die letzte Aktion erfordert das Hinzufügen des Widgets zum Sonar-Dashboard. Dies ist jedoch nicht notwendig, wenn das Plugin nur als Build-Breaker eingesetzt werden soll. Nach einer erneuten Messung wird das Widget im Sonar-Dashboard dargestellt (siehe Abbildung D.1).

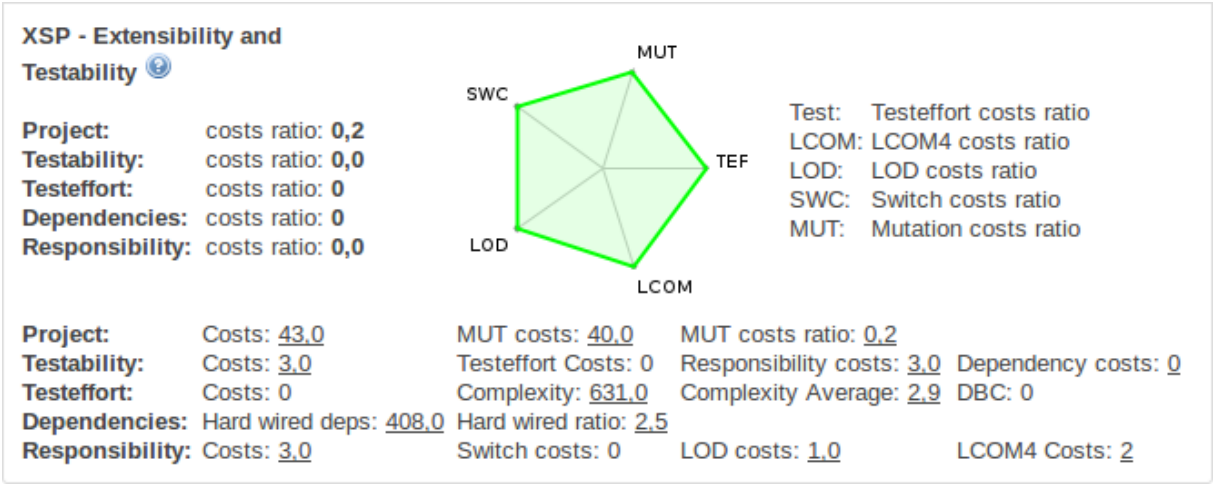

Abbildung D.1: Widget des Sonar-Plugins

Anhang E

Inhalt der CD-ROM

E.1 Latex-Sourcen

Der komplette Latex Sourcecode befindet sich im Ordner *thesis*. Entsprechend der Verwendung enthält dieser Ordner weitere Unterordner die selbsterklärend für sich stehen. Zur Kompilierung muss lediglich die Datei *masterthesis.tex* herangezogen werden. Alle anderen Dateien werden automatisch per Include in das Dokument eingefügt. Zur Kompilierung der Sourcen unter UNIX-Systemen ist folgende Vorgehensweise vorzuziehen:

Kompilierung in 3 Schritten:

1. pdflatex masterthesis.tex
2. makeindex masterthesis.nlo -s nomencl.ist -o masterthesis.nls
3. pdflatex masterthesis.tex

Anschließend ist im selbigen Ordner das neu kompilierte PDF-Dokument vorzufinden. Ebenfalls ist ein erfolgreiches Kompilieren nur dann möglich, wenn alle benötigten Latexbibliotheken installiert sind.

E.2 Sonar-Plugin

Das Sonar-Plugin ist als Quellcode und als Jar-Datei auf der CD-ROM enthalten.

E.3 Zusätzliches Material

- Patch für PMD-5.0
- Sonar-Pitest-Plugin
- Quellcode von Refactorings

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum Unterschrift